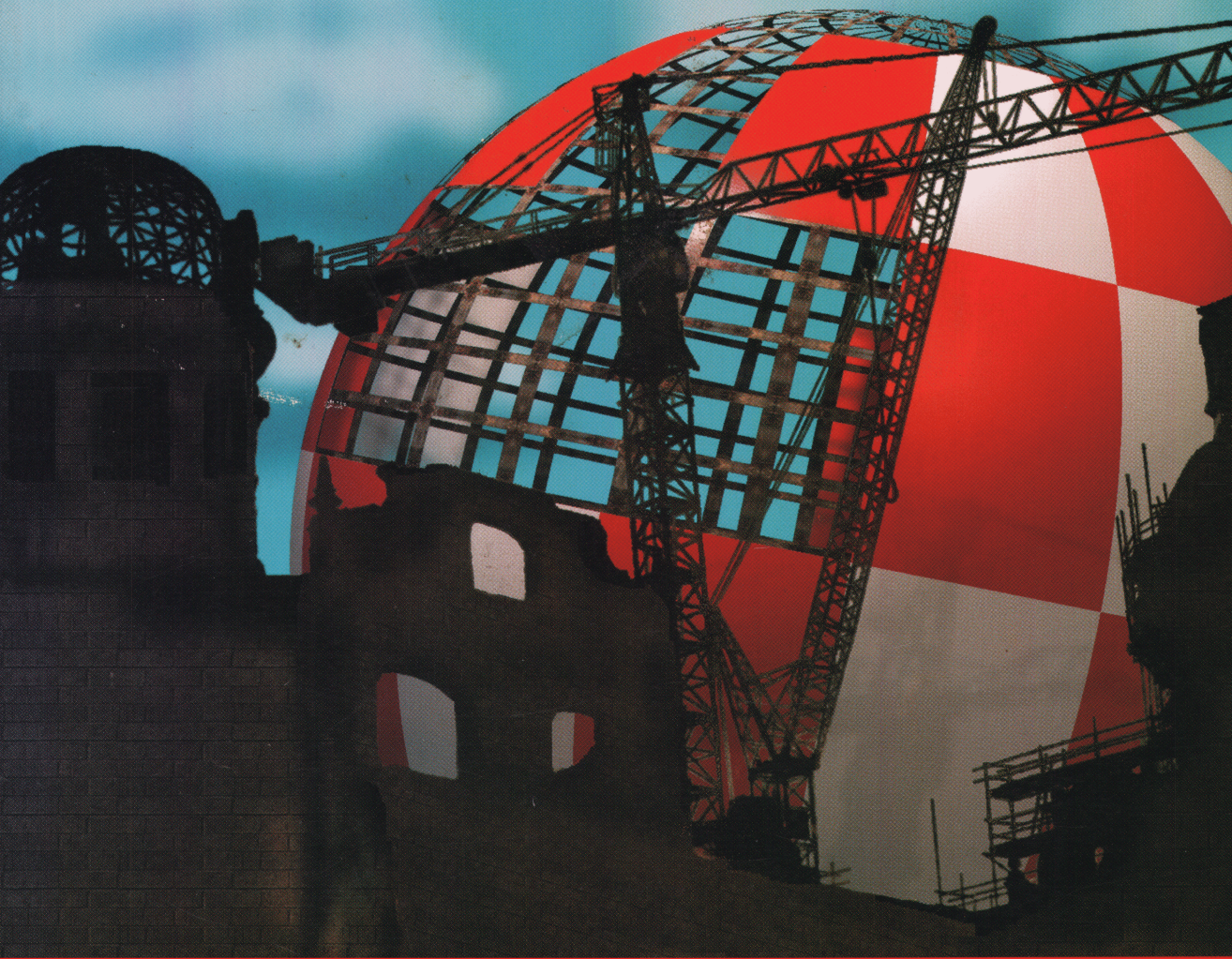




AMIGA[®]



Software Development Kit

Installation Guide

System requirements

- ❑ Red Hat Linux 6.1 running Xfree86
- ❑ 32 MB or RAM (64 MB recommended)
- ❑ 100 MB of hard drive space

To install the Amiga SDK

1. Start Linux and then Xfree86.
2. Mount the Amiga SDK CD
3. Run setup from the Amiga SDK CD.
4. Follow the instructions in the setup program.

Developer ID

You will be asked for a developer ID. It can be obtained from <http://www.amigadev.net/developer/register> or by calling 425-396-5640.

System ID

Once you have obtained your developer ID you will be asked for an unlock code. You can obtain your unlock code at <http://www.amigadev.net/developer/dbase/unlock.php> or by calling 425-396-5640.

Starting the Amiga OS

You may start the Amiga OS by typing `intent_shell` from a Linux shell prompt.

Contacting DEV Support

We encourage you to thoroughly research the developer web site at www.amigadev.net or to send e-mail to developer@amiga.com to answer any questions you have about the Amiga SDK. Once you have exhausted these resources and you still can't find what you are looking for call 425-396-5640 between the hours of 8:30 a.m. and 5:30 p.m. Pacific Standard Time.

Amiga Development

Second edition, May 2000

Edited by Simon N Goodwin, helped by Rudi Chiarito and Andrew Stout.

Developer support: Gary Peake. Product management: Matthew Fontenot.

Introduction

When this volume was proposed, the Amiga development team checked out all the documentation that they would like to include in printed form with the system shipped to third-party developers. It came to 1,500 pages, and even then there were further topics which we would have liked to document but which were too new and fluid to meet the deadline.

Rather than ship you a book that would be dauntingly difficult to read, and might be almost as thick as it was wide, we have condensed the details of the core of the new Amiga architecture into this (relatively) slim volume. This is not the whole story, but is enough of an overview to illustrate what we've got already, and where we are going.

Additional information about the Kernel of the system, the Shell and the Virtual Processor architecture is on your Developer CD, in PDF form, along with all the tools you need to read and print those documents.

Currently the Amiga OS is hosted on top of various platforms so that we can put it in your hands quickly. This allows you to begin developing tools and parts of the the new Amiga OS. Everything on the CD is geared towards introducing you to the VP language and helping you port your favorite tools to the Amiga platform. Good luck and enjoy!

Proprietary rights

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and the publisher was aware of a trademark claim, the designations have been printed in initial capitals. Amiga and the Amiga logo are registered trademarks of Amiga, Inc.

Copyright © 1998-2000 by Amiga Incorporated and contributors

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher.

Warning

Despite best efforts, the information described in this manual may contain errors and may not function as described. All information is subject to enhancement or upgrade for any reason including to fix bugs add features or change performance.

Disclaimer

Amiga Incorporated ("Amiga") makes no warranties, either expressed or implied, with respect to the information described herein, its quality, performance, merchantability, or fitness for any particular purpose. Such information is provided on an "as is" basis. The entire risk as to their quality and performance is with the user. Should the information prove defective, the user (and not their creator, Amiga, their distributors, nor their retailers) assumes the entire cost of all necessary damages. In no event will Amiga be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the information even if it has been advised of the possibility of such damages. Some laws do not allow the exclusion or limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitation or exclusion may not apply.

Realisation

This book was edited and formatted with Final Writer 5, Ghostscript, APDF and DPaint IV AGA on a Classic Amiga 4000 with a Cyberstorm 68060 processor.

Feedback

*We welcome feedback about this document. In the first instance this should be addressed by email to **developer@amiga.inc** Questions requiring a specific answer from our support staff should be addressed to **support@amiga.inc***

Table of Contents

Table of Illustrations

xviii

The New Amiga	1
Programming Environments	1
Software Infrastructure	1
Heritage	2
The VP Revolution	3
Amiga and Linux	4
The Amiga Loader	4
Loader options	5
Environment Variables	5
ELATE_ISINTERACTIVE	5
ELATE_KTRACE	5
ELATE_BREAKCHAR	6
Loader switches	6
Host Libraries	7
Interfacing to Native Filesystems	7
Debugging Amiga	8
xdebug	10
The Amiga Interface	10
Amiga Help Functions	10
Overview of the Amiga Shell	11
Command Conventions	11
Command Options	11
Simple Shell Commands	12
Listing directories with ls	12
Concatenate Named Files with cat	12
Benchmark current speed	13
Leaving Amiga via Exit and Shutdown	13
Directory Structure	14
Using Amiga Java	14
Simple Customisation	16
Virtual Processor Architecture	17
Components	18
Data types	18
Main Registers	20

Addresses	22
Protection	22
Alignment	23
VPcode program source	23
Addressing modes	24
Macro Instructions	25
High-level macros	26
Assembly-time macros	27
Jumping around	27
Tools	27
Object orientated calls	29
Conditional tests	29
Further reading	30
 VP Programmers' Quick Reference	 31
Instruction Types Description	31
Expression Types Description	31
Conditions Description	31
Abbreviations for Types	31
Instruction suffixes, formats and sizes	31
General purpose registers	32
Special purpose registers	32
High-level language macros	32
Assembly-time macros	32
Structures	32
Printf & Tracef formatting	32
Format Flags	32
Format Qualifiers	32
Format Conversions	32
 Introduction to Java	 33
Hello World	33
The Java Platform	34
Java bytecode	34
The Java Virtual Machine	34
The Registers	35
The Method Area	35
The Stack	35
The Garbage-Collected Heap	35
The Libraries	36
Object-Oriented Programming	36
Classes	36
Subclasses	37

Single Inheritance	37
Class Variables and Class Methods	37
Access Control	38
Packages	38
JIT, the Just In Time Compiler	38
Avoiding Constant Recompilation	39
Garbage Collection	39
Java and Embedded Systems	40
Real-Time Behaviour	40
Memory	40
The Amiga Java Platform	41
Small Footprint	41
Object-Based Programming	41
Amiga Dynamics	42
Real-Time Multithreading	42
Translating a Java Language Class	43
Translating into Tools	44
Running Non-Text Files	44
 Amiga Java Architecture	 45
Java Technology and Amiga	45
The Java Platform	46
The Java Virtual Machine	46
The Java Libraries	46
Virtual Machine and Library Testing	47
Transporting Amiga, Java Technology Edition	47
Objects and Amiga	48
Multithreading	49
Using Amiga Java	50
The Development Environment	50
Jcode Translation and Execution	50
Mixing Dynamic and Static Execution	50
Dynamic Run-Time Execution of Java bytecode	51
Running The Application	51
Static Build-Time Translation of Java bytecode	52
Translating Programs with Multiple Classes	52
Directory Mappings	52
Garbage Collection	52
The Amiga applet viewer	53
The class loader	54
Java archives	54
Java Beans	54
Unicode	56
Web connections	56

The Amiga C Compiler	57
Front End	57
Pre-processor	57
C to Assembler Translator	57
Object Files Generator	58
Linker	58
Labels	60
Assembler	60
Compiling C	61
Pre-processing	61
Translation	61
Linking	61
Assembly	61
Compilation Examples	62
Default Paths	62
Compiler Configuration and Implementation Notes	63
Data Formats	63
Register Usage	63
Function Call Conventions	63
Stack Usage	64
Debugger Support	64
Tracing Support	64
Tool Generation Support	64
Packed Alignment	64
Error Messages	65
Troubleshooting	65
exit() and taos_exit()	65
main()	65
Device Driver Interrupt Handlers	65
Porting existing C code	66
Data Type Conversions	67
errno	67
Compiler Options	68
The vpcc Command Line	68
Command line options	69
Default options and paths	73
Compilation examples	74
vpld Options	74
vpld pseudo-ops	76
Module Control	76
Linkonce blocks	78
Multiple modules	78
Module start	79

Declarations	79
Data Initialisation Pseudo-ops	83
Code Pseudo-ops	84
Diagnostics Pseudo-ops	85
Interface Pseudo-ops	87
Compatibility Pseudo-ops	88
More about Amiga C	89
Portability	89
Re-usable Tools	90
Greeting the world	91
Building and using tools with C	92
Declaring Tool Calls	92
Using Tools in C++	93
C++ to Assembler Translator	93
Creating callable tools in C	94
demo/example/c/mathtest.c	96
demo/example/c/mathtest.h	96
demo/example/c/usemath.c	96
Object-based style	97
Design issues	97
Memory Structure	98
Class include files	98
baseclass code	98
subclass code	98
Instance allocation and deallocation	99
Defining a Class and Method Coding	99
Example baseclass method framework:	100
Allocation and De-allocation of Memory	101
Allocating memory using the _new tool	102
Allocator code	102
Deallocating memory using the _delete tool	103
Deallocator code	103
Initialising and deinitialising instance data	103
Baseclass code	105
Defining a Subclass	107
Subclass methods	108
Subclass code	108
Calling an object from within an application	110
The named call	110
Baseclass test	111

Subclass test	112
File names and paths	114
Format of listings	114
A Classy example	115
Node class include file	115
Node class code	115
Job class include file	116
Job class code	117
List class include file	118
List class code	118
Queue class include file	120
Queue class code	121
Developing Tools	123
Assembling a Program or Tool	123
Hello to ASM	123
Example Tool Source	123
Structure of Application Source Files	124
Include Files	124
The Primary Tool	125
The tool name macro	125
The Assembler Language	125
The Tool Flags	125
The Stack Size	126
The Global Variable Size	126
Example uses of the tool macro for primary tools	126
The toolend macro	126
Accessing Command Line Parameters	126
Tidying up before closing a tool	127
Non-primary tools	127
Inside the Virtual Processor	128
Register Files	128
Integer Registers	128
Pointer Registers	128
The ent directive	128
Special Entry directives	129
VP Instructions	130
Instruction basics	130
Return of the Guru	131
Expressions	131
qcall, go, gos and ncall	132
qcall	132
ncall	133
go and gos	133

Labels or Tags	134
Structures and Memory Allocation	134
Global Variables	135
Local Variables	135
Allocated memory blocks	136
Bit structures	137
Program Control	138
Loops	138
while .. endwhile	138
for .. next	138
repeat..until	138
loop-endloop	138
Conditional code	139
if-elseif-else-endif	139
switch-whencase-otherwise-case-endswitch	139
bool macro	140
Rolling your own Macros	140
Register Defines	141
Tracing	142
Trace device	142
Tracef	142
ktrace.log	143
Error checking macros	143
Linked list macros	144
General purpose macros	146
Using Libraries	146
Changes to an Application Source File	147
Changing the tool definition	147
VP or native Tool Selection	148
Special Registers	149
The Stack Pointer, SP	149
PP demonstration	150
The Link Pointer, LP	150
The Global Pointer, GP	150
 The Amiga Kernel	 151
Kernel Overview	151
Kernel Features	151
Kernel Services	151
Tool Management	152
Non-virtual Qcalls	152
Virtual Qcalls	152
Virtual+fixup Qcalls	153
Tool management calls	153

Process Management	154
Process creation and deletion	155
Process creation helper functions	156
Process control	156
Process scheduling	156
Process parameters	156
Spawning	156
Scheduling	157
The PID or Process ID	157
Inter Process Communication (IPC)	158
Counting Semaphores	158
Mutexes	158
Event Flags	159
Mailboxes	159
Synchronisation Groups	159
Memory Management	160
Memory objects	160
Allocator classes	160
Other useful tools	161
Timer Management	162
Interrupt Handling	162
Signals	163
Signal actions	163
Signal functions	164
Sets of Signals	164
Callbacks	164
Named Data Areas (NDAs)	165
Atoms	165
Static atoms	165
Dynamic atoms	166
Atomic Linked List Functions	166
Kernel Device Functions	166
Kernel Entropy Collector	168
Kernel Time Functions	168
Ebug Technical Architecture	170
Other Utilities	171
DFA	171
Using DFA	171
DFA test categories	172
TCA	172
Using TCA	172
Instrumenting Object Tools	173
Running The Test	174
Hints and Tips for Programmers	175
Using the Developer Documentation	175

Tracking Which Tools are Loaded	175
Using the Amiga debug device driver	175
Translation Procedure	176
System overview	177
Amiga Modules	177
Core Subsystems	177
Libraries and Toolkits	178
Development tools	178
Block Diagram	178
Platform Isolation Interface	179
PII Services	179
Kernel services provided by the PII	179
Device driver services provided by the PII	180
Kernel and device driver services provided by the PII	180
PII Tools	180
System Startup and Shutdown	180
Information Provision	180
Memory Allocation	180
Memory Management	180
Interrupt Management	181
Exception Management	181
Host system specific	181
Process Synchronisation	181
Diagnostic Functions	182
Device drivers	182
Introduction to device drivers	182
Writing Device Drivers	183
Memory Mapping	183
Access to Input/Output (I/O) Ports	183
Exclusive Software Resource Access	183
Page Out Prevention	183
Device Driver Read/Write Policy	184
Device Driver Methods	184
Device Driver allocator and de-allocator tools	185
Workings of an _init Method	186
Workings of a _deinit Method	186
Workings of a read Method	187
Workings of a write Method	187
The Interrupt Service Routine	187
Blocking, Non-blocking and asynchronous policies	188
Non-blocking	188
Blocking	188
Asynchronous I/O	188
Notification mechanism	189
AIO structure	189

Interrupt service routine	189
Sequence of operations for asynchronous I/O	189
Device driver method implementation	190
Example code	190
Application	190
Callback handler	190
Interrupt service routine	191
Extensive interrupt processing	191
Starting and Stopping Device Drivers	191
Using devstart	192
Loading Device Drivers From Software	192
Accessing Drivers From an Application	193
The open method	194
The close method	194
Device Families	195
Available Device Drivers	196
Device Driver Types	196
Classification of drivers	196
Generic Device Drivers	197
NULL device (character)	197
Trace device (character)	197
Keyboard cooker device (character)	198
Tool loader	198
ZIP file system	198
File block driver (block)	198
Partition driver (block)	198
Character link (link)	198
File name mapper (file system)	198
PPP (network)	199
Modem (network)	199
FAT file system (file system)	200
Messenger (messenger)	200
TCP/IP (protocol)	200
Merge file system (file system)	200
PCI	200
Block cache driver 2.0	200
Amiga filesystem (filesystem)	201
Layered keyboard (keyboard)	202
Layered mouse	202
Layered pen	202
Display Device Drivers	203
Graphics Displays	203
Text Displays	203

Using The Amiga Shell	207
Command Options	208
Example Commands	208
Listing Files with ls	208
Concatenating Files with cat	208
Changing Directory with cd	209
Printing the Working Directory with pwd	209
Creating New Directories with mkdir	209
Leaving Amiga with Exit or Shutdown	209
Available Editors	210
The Shell Line Editor	210
Emacs keymapping	212
Key Bindings	213
ED	213
Regular Expressions	213
Addresses	214
Ranges	214
Commands	215
Search and replace	218
Search and replace flags	218
ED Options	220
JOVE	220
Invoking Jove	220
Jovial Help	221
Jove Options	222
Shell Interaction	222
Shell command synopsis	223
Shell command options	223
Advanced Shell Usage	224
Multitasking Functions	224
Process status	224
Running Shell Scripts	225
Shell Variables	225
Variable names	226
\$ Expansion of Environment Variables	228
Filename Generation (Globbing)	229
Pattern Matching	230
Redirection	231
Exception throwing	234
Exception catching	234
Notes on Catching	235
Shell Grammar Reference Guide	236
EBNF syntax	236
Command lists	236

Commands	236
Simple Commands	237
Redirections	237
Arguments	239
White space	240

Multimedia Toolkit 241

Conceptual Overview 242

Audio Visual Objects	242
AVE Device Event Handling and Tokens	243
AVE Device Calls	245
Tools dev/ave/tao/lock and dev/ave/tao/unlock	245
Method opentoolkit	245
Method closetoolkit	245
Method nextid	245
Methods allocevent and freeevent	245
Method aveinfo	245
Tool dev/ave/tao/script	245

Scripts and the System Menu 246

System Applications	246
Input device system applications	246
Utility system applications	246

Audio Visual Object Event Handling 248

Handling Events	249
Modality	249

Standard Audio Visual Objects 250

Application Audio Visual Objects	250
Gadget Audio Visual Objects	250
Window Audio Visual Objects	251
Pixelmap Audio Visual Objects	251

DSFX and GRF 252

GRF	252
DSFX	252
Driver files	252
Connections	252

Directory Structure 253

AVE Programming 254

AVE Program Structure	254
Blend Example	254
Event Types	254
System Events	254
Mouse Events	255
Comms Events	255
Token Events	255

Keyboard Events	255
User Events	255
Event Message Format	255
AVE message structure	256
Event Linking	257
Event linking methods	257
An event handler loop	258
Gadget Programming	259
Gadget Types	259
Toolkits and Properties	260
Anatomy of a System Application	261
Key code conversion	262
Keyboard system code	262
Open keyboard driver for AVE use	264
Get event from driver	264
Synchronous read	264
Device data	265
Creating New Audio Visual Objects	266
Displaying Audio Visual Objects	267
Contexts	268
Sizing and Layouts	268
Pix and Img Programming	268
Pixelmap Audio Visual Objects	268
Colour Encoding	269
Alpha	269
Blitting and Copying	269
Drawing Primitives	269
Pixelmap Types	270
Xmethods	270
Blitting and Copying	271
Mixing the trick	271
Image Audio Visual Objects	272
Sounds	272
DSFX Device Driver	272
The Amiga Font System	274
Import/Export Utilities	276
Loading	276
Saving	277
Converting Classic IFF formats	277
Audio Visual Object bitmasks	278

Glossary

279

AIFF	279
Alpha	279
ARGB	279
AU	279
AVE	279
AVO	279
Beans	279
Blitting	280
BMP	280
Bytecode	280
CISC	280
Class	280
Classname	280
Data hiding	281
defaultmethod	281
DSFX	281
Dynamic Binding	281
Elate	281
Encapsulation	281
Event	281
Font	282
FPU	282
Gadgets	282
Glyph	282
GIF	282
GRF	282
GUI	282
IEEE-754	282
IFF	283
Inheritance	283
Instance	283
JPEG (JPG)	283
Kerning	283
Latency	283
Linux	284
Mailbox	284
Method	284
MPEG	284
MP3	284
NCALL	284
Node	285
Object-based programming	285
Objects	285
PDA	285

PPC	285
Panose	286
PID	286
Pixelmap	286
PNG	286
Polymorphism	286
PostScript	286
QCALL	287
RISC	287
Tao	287
TGA	287
Thread	287
Tool	287
TrueType	287
Typeface	288
Unicode	288
Virtual Processor (VP)	288
VLIW	288
VPcode	288
WAV	288
xmethod	288
XOR	288

Index **289**

Table of Illustrations

Directory paths	14
Quick Reference card	32
Unicode UTF-8 bit packing	56
C compiler data formats	63
C compiler register usage	63
The Baseclass	105
The Subclass	107
VP supported processors	148
Stacks grow downwards	158
Process states	154
System block diagram	178
Kernel and PII connections	179
Supported device families	195
Types of device driver	196
The file name mapper	199
File hosting by block access	201
Shell background process	224
Multimedia sub-systems	241
How events are dispatched	243
Pointer and joystick events	247
Three levels of events	248
AVE connections	252
AVE Message structure	256
Input sequence	261
The raw and the cooked	262
Deriving a button gadget	266
Creating a new gadget	266
Object-based concepts	285

The New Amiga

This document is an overview of the Amiga development system, with particular effort directed at describing the foundation layer provided by the Tao Group. This chapter introduces the main components of the software. There's yet more detailed information on the accompanying CD, in PDF and HTML form.

The remainder of this manual is organised in four substantial chunks: one for each of the three programming environments, C, Java, and VPcode, followed by one discussing the operating system foundation.

Programming Environments

Of these chunks the first three are most important to new developers: C for porting existing code, Java for writing new portable applications, and VPcode for addressing the Amiga system directly. You need to learn about these, and the unique aspects of their Amiga implementation, because that knowledge is fundamental, and will stand you in good stead for years to come.

Software Infrastructure

The fourth section of this manual discusses the existing infrastructure of the system - the tool and object system, the Audio Visual Environment, and the command shell.

Those parts of the system are the most likely to change in future releases. They are documented now, because they are a foundation for the development of usable applications and utilities.

This is the first release of a system, and it is currently intended for developers, rather than end users. It is for those who want to create tools and applications for the new Amiga OS. This SDK is a starting point for those who wish to be a part of Amiga's future.

Later systems will run the Amiga operating system natively. They will be tailored to particular markets, with just the hardware they need. Interfaces, ports, even the processors will change, but the Amiga core will remain.

Heritage

Many of the concepts in this system will be familiar to Classic Amiga developers. The biggest change is the move from a system tied to one hardware architecture and processor, to one that can be scaled almost indefinitely; one that can take advantage of the latest hardware, yet release developers from the need to rework their products with each update.

If you're familiar with the old Amiga, there will be some things that you've grown used to, which are currently missing. We can already hear some of you crying out, what's happened to Rexx, or IFF, or BPLCON?

We're expecting you to help us flesh out some of these aspects of the new system. All these Classic Amiga features - yes, even BPLCON3 - can be implemented on top of our development system, and made available on new production machines.

The developers at Amiga Inc. understand the importance of the Amiga's heritage - it put us where we are today. The question is one of priorities - which parts do we need first, and which can we live without, in the short term, or longer?

While our attention is focused on the new aspects of the Amiga system, you know best the classic features you need. Once you're absorbed the architecture, you'll see how it fits together, and old and new parts can be integrated.

New Amigas combine the best of the Classic with features that could never be added, for fundamental reasons which made sense in 1984 but are obsolescent today. We are no longer tied to one processor architecture, or one chip set.

Indeed, we are not tied to a single processor - new systems may have various chips, or multiple processors, passing messages and dynamically allocating work to the chip best suited for a given job. This is something that the old AmigaOS could never have managed, and it has required some re-thinking.

We've chosen an architecture that gives you freedom, yet supports you, making it relatively easy to build up sophisticated solutions without wasting effort. Now is your turn to help shape things, as you may have done for the Classic Amiga.

The VP Revolution

The new Amiga's revolutionary operating system can be used on anything from mobile phones to high end servers. This complete portability is brought about by the Virtual Processor for which all Amiga programs are written, without regard for the hardware platform.

Once a simple program called the Translator is written for the new architecture, all application software and libraries written for the virtual processor can run immediately upon any platform or processor, with absolutely no need for any further rewriting or recompilation. As a result of this Amiga constitutes an effective Operating System across the broadest range of processors - be they CISC, RISC, VLIW or whatever else the silicon foundries may yet dream up.

Amiga's unique translation technology takes the Virtual Processor byte code and translates it into the native code of the target processor. Normally, translation into efficient native code only takes place when loaded from disk or network, rather than at compile or link time. The translator knows which processor it is running on and can generate the appropriate code. Programs for Amiga can currently be written in VPcode, the assembler language of the virtual processor, C, C++ or Java, with more compilers and interpreters under development.

Amiga incorporates the following features:

- Total portability
- Minimal footprint
- Redefinable realtime kernel
- Multitasking and multithreading
- Parallel and heterogeneous Processing
- An unique yet standard Java implementation

In addition, satellite products based around Amiga are available, including:

- Shell
- Debugger
- VP Assembler
- Disassembler
- C Compiler
- C++ Compiler
- Example Programs

Amiga and Linux

This section covers the basic details of installing, configuring and using the Amiga development system. Amiga is temporarily hosted by the Linux system. Further technical information is available in the form of online help, in formatted user guides, or from Amiga's Technical support group support@amiga.com.

The Amiga Loader

The Amiga loader is a Linux program which forms the virtual hardware for an Amiga session. It provides the interface between Amiga and Linux system calls.

To start an Amiga session, run this command:

```
sys/platform/linux/elate
```

from the root of the Amiga tree. For example:

```
cd ~/elate
sys/platform/linux/elate
```

This command loads an Amiga image and runs it, thus starting an Amiga session. What the session does is determined by the sysgen file used to build the image, together with any Amiga shell command provided by the `-c` option. The current directory is used for two purposes:

The directory searched for plugin libraries is `sys/platform/linux/ix86` by default, or `/ppc` relative to the current directory. This can be overridden with the `-l` option. The usual development system 'sysgen' image mounts the current directory as the root of an Amiga native filesystem.

The normal configuration of Amiga under Linux uses the **gwc** memory allocator. In this case, Amiga will request memory from Linux in large blocks which Amiga manages internally. Amiga can also be invoked under the **ebug** debugger, or the **xebug** script which starts ebug and Amiga in separate X terminals.

By default, the loader will run the image file:

```
sys/platform/linux/f15.img
```

It is possible to specify which file to run using the `-B` parameter. For most development purposes, the `f15t.img` file is more useful; this uses the checking translator, which does stack checking and `ent` block argument checking and has several other debugging aids, which impose a slight but worthwhile speed penalty:

```
sys/platform/linux/elate -Bsys/platform/linux/f15t.img
```

Loader options

It is possible to modify the behaviour of the loader with optional switches. The format is:

```
sys/platform/linux/elate -[switch] [argument] ...
```

If you wish to run a single Amiga command, you can do this with the `-c` option. For example:

```
sys/platform/linux/elate -c "terminal"
```

For some Amiga commands it may not be appropriate to start an interactive session, which is done by default. You can prevent this from happening by setting the `ELATE_ISINTERACTIVE` environment variable before running Amiga. If you use **zsh** or **bash**, then this will do the trick:

```
ELATE_ISINTERACTIVE=0 sys/platform/linux/elate -c "uname -a"
```

in this example, **uname** stands for an arbitrary command of your choice.

Environment Variables

These are the relevant environment variables, using the system's internal name.

ELATE_ISINTERACTIVE

When set to 0, disables the terminal interface. This is intended for when you're using batch jobs, rather than using the Amiga shell interactively. The default is 1.

ELATE_KTRACE

When set this redirects **ktrace** output to the specified file. The default is to send **ktrace** output to the screen.

ELATE_BREAKCHAR

This defines which character is used for the break character. Specify the ANSI code of the character desired. The default is 29, ^].

Loader switches

These switches may be added on the line that invoked Amiga. Switches are case sensitive and a space may be used between a switch and its argument.

`-B <imagename>`

Specifies the image file to load. The default is `sys/platform/linux/f15.img`.

`-c <shellcmd>`

Specifies an Amiga command to run. If this switch is used, then instead of starting the Amiga shell, the system will execute the command and exit. This can be useful for automated tasks.

`-h`

Help - Prints the usage message, rather than starting the Amiga system.

`-l <path>`

Specifies the path to use when loading dynamic libraries. Most users will not need to use this.

`-M <memsize>`

Specifies the initial amount of memory allocated for Amiga by the loader. The default is 4MB. A space may be used between the switch and the argument.

`-q`

Quiet - don't print the banner on startup.

`-U <incsize>`

Specifies the amount of additional memory allocated from the system when Amiga runs out of memory. Whenever Amiga runs low, the loader will allocate this much more memory. In common with other Unix systems, memory is not freed when it is no longer used. The default is 4MB. The increment unit must be larger than the largest size requested in a single allocation.

`-X <maxsize>`

Specifies the maximum amount of memory that Amiga is allowed to use. The Amiga process will never grow larger than this amount. The default is 64 MB.

Host Libraries

Amiga will generally require a certain set of libraries to be present on the host system. These include **libc6**, **glibc 2.0/2.1**, with the corresponding libraries for **libncurses** and the X Window System libraries. The required kernel version is 2.2.5 or later at the time of writing, though these numbers may change.

We have gone to a lot of trouble to compile a complete and compatible system for the developer boxes. You are advised to keep the system that we have shipped. If you can't resist changing the Linux environment, please be cautious and be ready to put things back the way they were if the Amiga system misbehaves later. If you do not heed this advice we might have to restore everything for you, or tell you how to do so, probably losing all the changes you have made. for better or worse.

Interfacing to Native Filesystems

The merge filesystem amalgamates several different filing system mount points into the merge device driver's mount point. If two or more filing systems share directories that have the same path, then the contents of these directories are concatenated together when accessed via the merge filesystem mount point.

The merge filesystem reads archives as if they were part of the normal file system, using the same drivers but calling the ZIP filesystem rather than the fat one.

If an attempt is made to modify a file within a read-only filesystem, the merge filesystem will make a copy of the file on the first writable filesystem available. This copy is used for all subsequent accesses to the file.

Debugging Amiga

ebug is a machine level debugger. It is particularly useful when starting up Amiga. Running the `f15t` image under it, the user can detect the following errors at run time:

- Incorrect parameters
- Misaligned loads and stores
- Running out of stack

The basic format of ebug invocation is as follows:

```
sys/platform/linux/ebug [<ebug_options>] <amiga_driver> [<amiga_options>]
```

For example:

```
sys/platform/linux/ebug sys/platform/linux/elate -Bsys/platform/linux/f15t.img
```

When starting Amiga under ebug, the current directory must be the Amiga root, as when running Amiga without ebug.

When some part of the Amiga system or any application it is running causes a fault, or you cause the Amiga session to receive a SIGINT, the Amiga system is stopped and ebug is entered.

It prints a brief description of the problem, then a register dump, then the source line where the problem occurred, as long as the tool was assembled with the `-g` option, followed by the name of the tool where the problem occurred and the offset into that tool, then a disassembly of the instruction which caused the problem.

When ebug is entered for the first time, it scans the Amiga system to find all the Amiga tools and atoms. If execution is continued, tools that are loaded and atoms that are allocated are remembered by ebug. A tool name becomes an ebug symbol like `dev/tool/elate/class`, and an atom name becomes an ebug symbol with a `#` sign on the front.

For example, typing

```
l dev/tool/elate/class+80
```

means *list (disassemble) instructions starting 80 (hexadecimal) bytes on from the start of the code of the tool dev/tool/elate/class.*

This lists some instructions. On an x86 machine, the first of those is:

```
00000213FF81 010ADDC0 cmp edi,00000213 (#_init)
```

This indicates that \$213 (hexadecimal) is the value of the atom `_init`. Similar results will appear on other platforms, though the machine code mnemonics vary between processors.

Tool and atom values can be converted using the `? instruction`:

```
type ? dev/tool/elate/class
result 010ADD40 (dev/tool/elate/class+0)
type ? 010ADD40
result 010ADD40 (dev/tool/elate/class+0)
type ? #_init
result 00000213 (#_init)
type ? 213
result 00000213 (#_init)
```

For more information about ebug please see the documentation on the Amiga CD.

xebug

xebug is a simple wrapper script for the ebug debugger on Linux. It should be invoked in exactly the same way as ebug. The only difference is that it spawns a new xterm window which will be used for the Amiga session's input and output.

```
sys/platform/linux/xebug [<ebug_options>] <amiga_driver> [<amiga_options>]
```

Input to and output from the debugger addresses the original terminal window. This means that output from the program under test is not mixed in with debugger prompts. That's particularly convenient if the program being debugged uses full-screen output and cursor addressing.

The Amiga Interface

Once Amiga has been installed and started up, an Amiga session may be run within a terminal window.

Amiga Help Functions

Online help within the Amiga system is provided in standard HTML format, and can be accessed through any external browser. Online help is also viewable. Two functions are particularly useful in this respect. They are **help to** view a help file, and **html to** view an arbitrary file written in the hypertext markup language.

The syntax for the help command is

```
help <text>
```

For instance:

```
help grep
```

The documentation for the shell command grep appears within the Amiga shell.

An index of all Amiga documentation available online can be found within the Amiga root directory. The documentation is listed according to category, (i.e. device drivers, shell commands and so on). To view this file, type:

```
html contents.html
```

Overview of the Amiga Shell

The Amiga Shell is a scripting command language interpreter. It is able to read and execute commands from the user, and thereby provides an interface to the underlying Operating System.

The commands you enter are dealt with by the command processor, which calls on the services provided by the Amiga kernel. The results of most commands are sent to standard output, which is typically the screen. The Amiga shell has a similar feel to standard Unix shells; although it's designed to offer a level of functionality comparable to a **zsh** shell, it has a much smaller footprint.

Commands understand and use familiar concepts of redirectable standard input, standard output, and standard error (for reports) like Classic AmigaOS or Unix.

Command Conventions

Commands can be entered at the \$ prompt, displayed on the screen within the Amiga shell. Where a command line is shown, text enclosed in angle brackets **<thus>** should be replaced by an actual parameter when typing a command. Parts of the command line shown in square brackets **[thus]** are optional.

So for example :

```
command <parameter1> [<parameter2>] [<parameter3> ...]
```

In this case parameter 1 is mandatory, and parameters 2 and 3 are optional. However, parameter 3 cannot be specified unless parameter 2 also is. Parameter 3 may be repeated.

Command Options

Commands can be modified by specifying additional options. All options must be preceded by "-". Multiple options can be specified together, so for example, "-abc" would be treated the same way as "-a -b -c". However some options may take additional arguments, in which case multiple options should not be specified.

Options can appear anywhere on the command line, between parameters. For clarity it is recommended that options be placed immediately after the command name, and therefore before any parameters.

Simple Shell Commands

Here are examples of some simple commands with their approximate equivalents on older systems. All of these commands can be run from the command line:

Operation	New Amiga	Classic	Unix	MSDOS
Benchmark test	speed	n/a	n/a	n/a
Concatenate files	cat	type, join	cat	type
Copy arguments to standard output	echo	echo	echo	echo
display date	date	date	date	date/time
Exit shell	exit	endcli	exit	exit
List directories	ls	list, dir	ls	dir
Quit Amiga	shutdown	CtrlAmAm	shutdown	CtrlAltDel
Run Java class	jcode	n/a	java	java

Listing directories with ls

This command lists the named files. If a directory is named, its contents are listed. If no file names are given, the contents of the current directory are listed instead. Typing this in after the command prompt:

```
$ ls [<filename> ...]
```

would produce something like this:

```
dev      ebug.exe   feq.exe    lang      makefile
app      docn
```

Concatenate Named Files with cat

By typing in something like this:

```
$ cat [<filename> ... ]
```

it is possible to either place the content of any named files in a new file, or copy the input typed at the keyboard to a file, using redirection.

Benchmark current speed

The **speed** command benchmarks the speed of the current processor, and displays the results to standard output. The speed command results in something like this:

```
VP MIOPS = 62.060606 (integer)
VP MLOPS = 0.969696 (long)
VP MFOPS = 0.290909 (float)
VP MDOPS = 0.028985 (double)
```

The rows show how many millions of these fundamental integer, long, floating and double operations can be performed every second.

Leaving Amiga via Exit and Shutdown

`Exit` terminates the shell, while `shutdown` shuts down Amiga. These have almost the same effect from the initial shell, but nowhere else. It is strongly recommended that this command should be used to shutdown the system, to ensure that proper tidying up is done.

Directory Structure

Files within Amiga are organised into directory structures - as is commonly the case, a directory is itself a file, containing the name and locations of the files it contains. Consequently any commands that apply to files are also applicable to directories. One or more files can be specified. These are the key directory areas:

Applications	app	All applications
AVE	ave	Multimedia toolkit
Com.uk	com	Java classes, using Java namespace conventions
Demonstration	demo	Example programs
Device Drivers	dev	Device drivers
TCP/IP Subsystem	etc	TCP/IP network and host configuration files
Fonts	fonts	TrueType and PostScript Type 1 fonts
Home	home	The user's home directory
Java	java	Java libraries
Languages	lang	Programming languages
Libraries	lib	General library files
Sounds	sounds	Audio-specific parts of the multimedia toolkit
System	sys	System directory

You're most likely to find required programs by following this directory structure.

Using Amiga Java

Amiga Java supports the PersonalJava 1.1 libraries. The following command is used to load and if necessary translate a Java class, so for example to run the application `hello.class`, type:

```
jcode demo/example/j/Hello
```

For compatibility reasons, it is also possible to format this command as follows:

```
jcode demo.example.j.Hello
```

The method described below is used as an entry point to the class specified in the `jcode` command:

```
public static void main (String[] args)
```

The name given to the `jcode` command needs to be the absolute name of the file.

Amiga Java can work with any standard Java development environment that produces Java classes in the standard byte code format. It fully includes **javac**, which can compile Java source code into bytecode form. The interface is identical to that when using javac under the JDK, as if you typed this at the shell prompt:

```
javac <source>
```

for example:

```
javac demo/example/j/Hello.java
```

For our simple example class, the standard directory mappings for code locations are as follows, so that:

```
demo/example/j/Hello.class
```

should appear in the directory marked:

```
<root>demo/example/j
```

The Java source code should have the appropriate package statement in it. For our example this would be:

```
package demo.example.j;
```

Amiga currently supports the Personal Java 1.1.6 API.

Introduction to the Amiga Multimedia Toolkit

The aim of the Amiga Multimedia Toolkit is to provide a component based user interface; a toolkit which provides the flexibility and modularity to build any form of audio visual environment. This allows multimedia elements, such as an MPEG video player to have a standard programming interface which can be incorporated into any application.

With conventional operating systems with a Graphical User Interface [GUI] the limit of the functionality of the program is directly imposed by the restrictions of the GUI. By contrast the Amiga multimedia toolkit imposes no such limitations, and incorporating the ability for Amiga multimedia toolkit users to build their own gadgets out of the tools provided, - that is, the ability to build their own set of authoring tools - bypasses the problem of the predefined nature of the interface development kit conditioning the end result.

The final vendor solution reflects the requirements and restrictions of the application, and not those of the Operating System or its graphical toolset.

Because there are no restrictions on the usage of the Amiga multimedia toolkit, there are no limitations upon what can be produced by it. The advantages of this modular, object-based approach is that applications are not determined by the look-and-feel of the user interface. From these tool-sets they will effectively be able to build their own GUI and applications for PDAs, telephones, digital cameras, interactive television set-top boxes, workstations or any other form of product.

Simple Customisation

The file `dev/ave/auto.scr` contains a list of the applications that are to be automatically launched at start-up. For example, the tiled Amiga backdrop can be configured to run a specific script when clicked. In practice, any application could be started through these means. However, it defaults to the system menu program `dev/ave/dsk/runapp.scr`.

This simply lists the directory tree rooted in `/app/start/`; each script file found here is displayed as a menu item, while subdirectories become submenus. Thus, to add an option to the system menu, create a script file to invoke an application at the appropriate place in the `app/start/` directory tree. The script's filename, minus the extension, becomes the label.

Other simple customisation can be effected through the properties system. The look and feel of the Amiga multimedia toolkit is determined through this system. The required attributes are read from a plain text file. By default this file is `dev/ave/default.prp`. Within this file it is possible to set attributes such as colour for all gadgets, just for particular types of gadgets, or for gadgets within a particular application.

Virtual Processor Architecture

This chapter explains the low-level workings of the new Amiga. It is aimed at people who need to write fast and concise code, compiler and code generator writers, and those who like to understand systems from the bottom up. If your main interest is applications, you probably do not need to know any of this, though you may find it interesting.

Programs for Amiga are written in code for a Virtual Processor, and translated into machine-specific instructions as they are loaded into the memory of a physical processor.

This approach means that our programs can run on any processor which has a translator for VPCode. They run much faster than programs written in conventional intermediate codes like UCSD P-Code or Java byte code, because the translator optimises the native code for the exact system which will run the program, at the time it is loaded.

The translator can compensate for performance differences between notionally compatible processors. Previous systems have been stuck with code optimised for a certain model, with varying results on other systems, which might range from a sixteen bit 386SX without cache to a Pentium Pro capable of munching several instructions at once. Microsoft systems still contain code 'optimised' for the 16 bit segmented 8086 architecture, which bears little relation to Intel's current offerings.

The Virtual Processor can optimise code to make best use of coprocessors, primary and secondary caches, and main memory architecture. It is not limited to Intel variants, but can make best use of Cyrix, AMD or rival chips, with other strengths and weaknesses. More importantly, it can bridge the gap between processor architectures, supporting RISC chips as well as CISC ones, Motorola as well as MIPS, Arm or Transmeta, or whatever delivers the best bang for the buck in a given application.

Amiga allows several processors from different families to co-exist in a system. It even lets users add new processors as yet unknown, and run their existing applications and system software on the latest silicon. All that needs to change is the translator, to accommodate the new processor. Even that can be altered as new techniques become available, boosting the performance of existing software and hardware.

Components

The Virtual Processor takes advantage of the fact that the core features of modern processors have much in common, despite their differences in detail.

All the processors Amiga targets have support for 32 bit data. Some can double that up to work with 64 bits in one step, while others use combinations of instructions to achieve the same effect. Some blur the distinction between 32 bit data and address pointers, while they are synonymous for others.

Floating point arithmetic may use single or double precision, 32 or 64 bit format, with any mixture of hardware and firmware implementation of individual operations. You program the generic facilities of the Virtual Processor, and let the translator worry about mapping these efficiently onto the processor of the moment.

Data types

The Virtual Processor has five types of register - 32 bit and 64 bit (long) integers, double and single precision floating-point values, and 32 bit pointers. These registers are referred to by a prefix letter - I, L, D, F or P - and a numeric suffix counting from zero. Thus P0 is used much like A0 in a 68K system, I0 resembles D0.L, and so on. You don't have to worry about the exact number of real registers on a given system - just use whatever you need, and let the translator work out where to put them.

Eight bit byte and sixteen bit short data types are also supported in VPcode. You specify these with a .B or .S suffix - .B matches the Motorola 68K convention, but .S is the equivalent of .W in Classic Amiga code. Don't confuse this with .S in a Motorola branch, which marks an eight bit offset. It means short - sixteen bit - format in VPcode.

Floating point values conform to the same IEEE-754 standard used in Power PC, Intel and 68K FPU's - indeed, used almost everywhere except in vintage Microsoft BASICs and old DEC systems. Floating point precision will vary between systems, so you should not rely on results being exactly the same on all hardware.

For instance 68K FPU's use 80 bit extended precision internally in working out 64 bit results, while Power PCs are 64 bit throughout. Some systems try to be as precise as possible in handling denormalised numbers - values so small that they have a zero exponent, and one or more zeros in the most significant bits of the mantissa. Other systems treat such values as zero. On such hardware, the Amiga

must sometimes use emulation routines for denormalised number support, as this is required for Java.

All Amiga systems support rounding of values when they convert from floating point to integer format. Some operations round to the nearest integer while others (D2LT, F2LT, F2IT) truncate, discarding any fractional part.

Floating point infinities, overflows and unrepresentable values (known as NaNs, or Not A Numbers) use special case values with maximum exponent (255 in single precision, 4095 in double precision) and un-normalised values in the mantissa. The exact values for each case, and the cases recognised, depend on the host; their meanings can be checked in a system-independent way with the ORD and UNO instructions, which perform ordered and unordered tests respectively.

All Amiga systems can run floating point code, but some may lack floating point hardware, performing single and double precision decimal maths by software emulation as the IEEE libraries did on a classic Amiga with no FPU. The same programs will run on any system, but floating point emulation is typically a lot slower than direct hardware execution of these operations.

The Classic Amiga Motorola FFP format is not supported, but Amiga does support fixed point values as well as floating point ones - these are 32 bit values with a sixteen bit integer part, ranging from -32768 to +32767, and a 16 bit fractional part with a resolution of one part in 65536.

Fixed point is appropriate on systems that lack an FPU but need to keep track of fractions in a limited range. It's often used in graphics, to keep track of inter-pixel alignment, for instance. It may or may not be more efficient than IEEE floating point, depending on the host system. There are special fixed-point multiple and divide instructions, DIVH and MULH.

Addition and subtraction work just like 32 bit integer operations, though the interpretation of the results differs. Whole-number constants can be loaded into fixed-point registers with the default .i data size, if you shift them left 16 places, as in `42<16`, which would load the constant 42 in fixed point normal format.

All values are treated as signed - the 32 bit integer range is therefore -2,147,483,648 to +2,147,483,647. Overflow is not detected - if the result of a mathematical operation generates more than 32 (or 64) bits, only those bits are stored. The overflow, carry or borrow is lost.

When you load a byte or short value into a 32 bit register, the unused high bits are

zero. This converts the value from signed to unsigned. You can force sign extension of values with B2I or S2I when they are loaded. The translator tries to generate instructions which do this automatically on the host processor.

Main Registers

You can give registers more memorable names with `def` macros. `defp` defines pointers, from `p0` onwards, `defi` assigns names to integer registers from `i0` up, and so on.

When you've finished with a register you can mark its last use with a 'zap flag' to indicate that it's no longer needed. This flag takes the form of a tilde ("~" , ASCII code \$7E, 126 decimal) after the name. The translator takes this as a hint that it can reuse the corresponding resource. If you do not do this the data flow analyser will attempt to perform zapping automatically.

If a register is used more than once in an expression, you can't zap it there, because the translator will not necessarily have finished with it when it comes to the zapped reference. You need to explicitly zap it in a following program line.

It just so happens that the C compiler uses 32 of each type of register - `i0` to `i31`, `p0` to `p31`, `f0` to `f31` etc. - but you can have as many as you like in your own VPcode programs. The only limit is the amount of stack space.

Room is reserved to store each VP register value when it is not mapped to a native hardware register. Thus a reference to `i99` allocates 400 bytes of stack space, for `i0` to `i99`. Your programs will be more frugal if you allocate register number consecutively from zero.

If the return data from a tool, subroutine or `ncall` is not required this can be indicated with a tilde at the end of the call line. This tells Amiga not to bother returning parameters after the call, thus increasing speed and reducing stack overhead. For example:

```
qcall demo/example/testtool, ( i0 : i~)
```

Special registers

There are four special Pointer registers, with alphabetic names: SP, PP, LP and GP. These are the Stack Pointer, Link Pointer, Parameter Pointer and Global Pointer respectively, and correspond to the base addresses used for common structures in compiled programs. SI is a fifth read-only special register, which holds a S**I**gnature used internally.

Normally these special registers are modified by the system, not explicitly by the programmer, though their values can conveniently be read. You cannot zap these because they're always potentially useful, and you are not allowed to modify them directly - if you ignore this advice, your code will crash, sooner or later.

The Stack Pointer addresses memory for temporary results. As each value is stored on the stack, the pointer grows downwards, towards lower addresses. The stack pointer always contains the address of the most recent item added to the stack. As items are removed from the stack the pointer moves to higher addresses. The stack pointer is always aligned to the natural word size of the real processor, with padding bytes as needed to preserve this alignment.

The Link Pointer holds the return address from a subroutine. This is the place where code execution continues after tidying up the stack at the end of a routine. The Link Pointer should not be altered by the programmer because changes might clash with assumptions or optimisations performed by the translator.

The Parameter Pointer is a read-only register which is set to the value of the Stack Pointer just before control is passed to another routine. It is used to pass variable parameters which cannot conveniently be transferred in registers.

The Parameter Pointer addresses parameters placed on the stack before the call, so that tools and subroutines can accept variable numbers of parameters to tools/subroutines. Alignment rules mean that target systems may put varying amounts of data on the stack, padding for alignment. Some use a Link Pointer internally and may not put the return address on the stack at all.

The Global Pointer holds the address of an area of memory accessible throughout the current execution thread. It too should not be changed.

There is one other special register - the Signature Register. This is used internally in a hashing scheme that speeds up object orientated code despatching - it holds an integer which changes when you make method calls.

Addresses

Addresses in VPCode programs refer to bytes, like 68K programs and C pointers, but unlike BCPL BPTRs (thankfully) or the word-addressed Texas 320 DSP range. The least significant byte of a word comes at the lowest address - this is the little-endian scheme (named after the pseudo-religious wars in Swift's satire, *Gulliver's Travels*) which Intel use. It's different from the Motorola 68K or Power PC schemes, where bytes come most significant first, in the order in which you would read them in a hex dump.

The Intel scheme makes dumps harder to read, but simplifies computers that process long values piecemeal, as they can fetch the least significant bytes first and generate any carries before they move on to more significant bytes, which will have to take the overflows into account. It's been a long time since processors worked this way - most can cope happily with 32 or 64 bits at one gulp - but like most things Intel, it was a decision made a long time ago which they've been stuck with ever after.

Serious problems occur if programmers get byte order confused. Processor manufacturers don't help by mixing them up - some even use one format for constants inside program code, and the opposite for data values stored and loaded by that program! Unix gurus call this the Nuxi problem, and joke that 'Nuxi is not a trademark of TAT&'. This is one of the most trivial yet fundamental differences between programming Amiga and AmigaOS.

VPCode has to pick one representation, whatever the format might be inside the host computer, and its designers opted for little-endian byte order. You don't have to worry about this unless you're writing something like a DMA device driver.

The VPCode translator automatically shuffles bytes so that they can be addressed in the natural way for a big-endian system by inverting the two least significant bits of the little-endian byte address. This 'address munging' is only necessary when accessing sub-fields of the natural types, as when plucking a byte from an int, or an int from a long. Ints, longs, floats, and doubles are all stored in their natural form on both big and little-endian processors.

Protection

Address zero is guaranteed to hold zero. This was normally the case under AmigaOS, and a surprising number of programs relied on this fact, often because they went one step to far at the end of a null-terminated list! If a base address is

not set, in C or other languages, it defaults to zero, causing the buggy program to attempt to access low addresses when they use that base to grope around inside a structure.

For this reason VPCode defines addresses in the range -128 to +127 as illegal, and reports mistaken attempts to access them. Enforcer, CyberGuard and MuForce in Classic Amiga systems protect the first page of memory with the MMU, for similar reasons. Programs that peek - still worse, poke - around there are likely to be broken, and it's useful to have that pointed out to developers.

Alignment

Values should be aligned on natural boundaries. System memory structures are always allocated this way - 32 bit integers at an address evenly divisible by four, double precision decimal values at an address which is an integer multiple of eight, and so on.

There are CPY, LD and ST instruction variants which can pick up values from odd addresses when necessary to read or write byte-aligned streams of data, but this is usually far less efficient than aligned access. These instructions prefix an N before the type, such as CPY.ND or CPY.NI, LD.NI or ST.NL. If a value straddles a natural boundary it must be fetched in two steps, with the parts shifted and merged before it can be used. This is an overhead best avoided.

VPCode program source

VPCode programs are written as lines of text, like other assembly languages, but there is no fixed correspondence between the number of lines and the number of instructions generated. Programmers can write complicated expressions on one line, and leave it to the translator to generate the instructions that get the job done.

The translator knows about parallel adders, barrel shifters and other goodies that can combine several steps into one instruction. It also takes account of pipeline stalls and data latency to keep the target processor as busy as possible.

The most basic VPCode instruction is CPY, short for copy - this is the equivalent of LD in Z80 code, MOVE in Motorola 68K, DEC or PPC code. The instruction is read from left to right, unlike the assignment syntax of Intel or Zilog instructions, or the old CP/M PIP utility which used the sequence newdata=olddata.

In VPCode the source operand comes first, followed by the destination:

```
CPY 8,i2
```

stores the constant eight in integer register 2.

```
CPY i0,i1
```

assigns the value of i0 to i1, not vice versa.

There is no equivalent of the separate Load and Store operations in 8080, 6502, 6809 etc. Those cranky old eight bit processors could only cope with one address at a time, so it made sense to distinguish instructions on the basis of the direction data was going, or on or off the chip. VPcode can write a result and read several memory operands in one line.

You can write an expression in place of a constant or register name, by enclosing the calculation in parentheses (round brackets, like these). The usual four dyadic infix operations can be expressed in words - add, sub, mul or div - or with the conventional symbols + - * and /. If both operands are constants the result will be worked out by the translator, rather than recomputed at runtime. This 'constant folding' makes programs more efficient than would otherwise be the case.

There is no equivalent of the three operand instructions of the Power PC; one of the source operands is always replaced by the result. However you can get this effect by writing an expression, like this:

```
CPY {i0+i1} ,i2
```

Addressing modes

Constants stand for themselves in VPcode. Thus `CPY 4,I0` puts the constant value 4 in the register. This is not like 68K code, where `MOVE 4,D0` would move the contents of memory address 4, and `MOVE (or MOVEQ)#4,D0` would be needed to assign the constant, signified by the hash # prefix.

In VPcode, you'd get the contents of address 4 (if it were a legal address, which it isn't!) with `CPY [4],I0`. The square brackets denote indirection, showing that the value is an address rather than constant data.

All the other 68K addressing modes, and more general cases besides, can be built up by putting expressions in and around the square brackets. `MOVE.B (A4),D7` becomes `CPY.B [P4],I7`, `MOVE.L 2(A0,D0),D1` becomes `CPY [2+P0+I0],I1` and

so on.

The constant 2 can be any size up to 32 bits, whereas the original 60000 limited the offset to an eight bit signed value. Integer registers are always 32 bits wide, in VPCode, so there's no direct equivalent of .W in MOVE.L 2(A0,D0.W).

The size suffix, .B is only a hint to the assembler. A common mistake is to write CPY.B I0,I1 and assume that only the bottom eight bits will be copied across. A full 32 bit copy is performed between the registers, and the .b is ignored because there is no memory access. Likewise if you write IF.B I0==I1 a full 32 bit comparison is made.

The destination does not have to be a register - it can also be an indirect reference to memory. Double indirection is possible, as on 68020 or later processors, where the expression yields the address of a location that holds the effective address for the CPY.

As with most machine languages, there are some limitations on the use of these modes - a few instructions require register, rather than memory, operands. These cases will be explained in detail in the online VP documentation.

What you don't get for free is autoincrement and autodecrement modes, a speciality of DEC and later Motorola processors. You have to use ADD, SUB, DEC or INC on another line and let the VPCode translator work out if the host processor can combine those into the addressing instructions it uses.

Macro Instructions

Some lines may generate an arbitrary amount of code, or none. These 'macros' correspond to pseudo-operations or the predefined compound meta-instructions of macro assemblers. Preset macros mark the start and end of blocks of code and assign memorable names to registers or constant values.

Macros let you express common operations in conventional ways. Rather than write everything as a CPY, such as:

```
cpy (i0 - 1),i0
```

you can write

```
sub 1,i0
```

The code generated is the same, either way. The second form is a macro which

generates the first line. The translator will generate a SUB or DEC, or some other instruction, depending on the requirements of the target processor.

A lot of familiar instruction macros are defined for comfort and convenience, such as inc, dec, add, sub and mul. Use these if they help you feel at home, but do not forget the power of letting the VP assembler work things out for you.

It may seem momentarily cool to write MULTIPLY WS-NUMBER-OF-LINES BY WS-PRICE-PER-LINE GIVING WS-INCOME-EARNED in a COBOL program, but once the arithmetic gets a little more complicated coders soon reach for the COMPUTE verb!

High-level macros

Pre-defined macros in VPcode can express high-level language constructs, like loops and conditional tests. They can also be used to allocate data structures and format output for debugging and diagnostics. The result is that you can write assembly source with the eloquence and clarity of a block-structured language, yet the low-level control of virtual machine code, and no need for representational tricks at either end of the spectrum.

Loops of code that are to be executed at least once can be written with REPEAT..UNTIL condition. Loops that may be executed zero or more times are written WHILE condition..ENDWHILE. Loops that execute a fixed number of times are written FOR count, register..NEXT register, where register is an integer register used to keep count of the loop iterations. BREAK, BREAKIF, CONTINUE and CONTINUEIF allow conditional or unconditional transfers to the outside of the loop, or back to the start.

Conditional execution may be encoded with IF condition..ELSEIF condition..ELSE..ENDIF or BOOL condition,label.

PRINTF, TRACF and KTRACE allow formatted output of text and variable values in binary, octal, hex or decimal integers, single characters, strings or formatted floating point values. The formats are summarised later, in the *VP Programmers' Quick Reference*.

ALLOCSTRUCT and FREESTRUCT simplify allocation and release of space for structured data. These macros, and more besides, are explained in more detail in the `VPmacros.pdf` file on your Amiga development system CD.

Assembly-time macros

If you want to include lines conditionally or insert code from other files when your VPCode source is assembled, you can direct the assembler to make insertions or decisions on your behalf. This is convenient when generating code for a range of systems, differentiating debug and production code, and to minimise the risk that you'll end up with lots of files with slightly different versions of a program, failing to keep them all up to date.

These macro instructions are all prefixed with a dot, to differentiate them from conditions and macros which will be interpreted later, when the assembled code runs. They include `.IF condition <code>`, `.ELSEIF condition <code>`, `.ELSE .<code>`, `.ENDIF`, `.INCLUDE <file>`, `.ALIGN`, and most significantly `.MACRO <code>` `.ENDM` which lets you define your own macros.

Jumping around

The VPCode unconditional jump instruction is `go`, like `JMP`, `JP` or `GOTO` in other languages. The smallest possible program is therefore:

```
Loop:  go Loop
```

Subroutines are called with `gos`, which is rather like `JSR` or `CALL` or `GOSUB` except that you can specify parameters after the destination label. These 'actual parameters' are copied to the 'formal parameters' in the corresponding `ent` line.

The address can be a value in a pointer register, rather than a label in the same tool. Thus `GOS P0` gives the equivalent effect of using procedure or function parameters in Pascal, for example. It may be a convenient way to call routines in other tools, having looked up their addresses at agreed offsets in a shared table.

Tools

VPCode programs are referred to as 'tools' made of 'entblocks', each of which is an independent subroutine. The first line of an entblock is `ent` (surprise!) followed by the name of the parameters, or a minus sign if there are none.

`QCALL` is the normal way to call code in another tool. An extra optional parameter at the end can indicate that the tool is to be loaded temporarily, which saves memory at the expense of time, or loaded on first reference and then

retained in case it's used again. This is rather like a call to an Amiga shared library except that the library is opened for you when required, and may be automatically flushed afterwards.

Internally, QCALL works by looking up tools by name and jumping to the corresponding entry point. This simple example shows the code both sides of a QCALL which finds the absolute value of a double-precision number in a register:

```
qcall lib/abs,{d0 : d1}
```

where lib/abs could contain

```
tool "lib/abs"
ent d0 : d0
if d0 < 0.0    ; -ve ?
neg.d d0      ; make +ve
endif
ret
```

In this case the caller can have both the original number and the guaranteed positive value by writing:

```
qcall lib/abs,{d0 : d1}
```

Whereas if only the absolute value is required, the programmer can write:

```
qcall lib/abs,{d0 : d0}
```

The VPcode translator automatically keeps a copy or re-uses the register, depending on the format of the call, not that of the tool.

You can even pass constants and expressions as parameters, like this:

```
qcall lib/abs,(-1.0 : d0)
qcall lib/abs,(d0 + -1.0 : d1)
```

Remember to include the brackets and separate the parameters with commas.

Object orientated calls

NCALL, PCALL and CCALL are low-level ways to perform object orientated calls.

NCALL invokes a method on an object. The first parameter after ncall is a pointer to the object, followed by the name of the method, then any parameters. If the method is not found in the object, the object's default method is called; typically this passes the call back to the object's parent class, and so on until it is recognised.

CCALL is similar but attempts to invoke a method on a particular class of an object. Consequently the first parameter is a pointer to the class tool, rather than to the object.

PCALL is used within a class tool, to invoke a method in its parent class. There is no pointer at the start of the parameters, as it always refers to the parent class. The first parameter is the method name, followed by the usual list of parameter registers, in brackets, if required.

Conditional tests

Programs don't achieve much unless they make decisions, so there are conditional instructions in VPCode. The implementation is different from that in conventional assembly language, for good reasons that reflect recent changes in fast processor design.

Conditional tests are a potential bottleneck on most modern processors, because they disturb the pipeline that feeds a steady stream of instructions to the execution units - when instructions are decoded and executed piecemeal, a condition causes a stall because the processor is no longer sure which path to follow to prepare further instructions for execution - it depends on the condition, which won't be evaluated till the associated instruction reaches the end of the pipe.

VPCode tries to get around this by asking the programmer to indicate whether a given branch is 'likely' to be taken, or not. It also does not implement any of the 'condition codes' familiar from older processor architectures, because they introduce architectural dependencies which would limit the speed and portability of VPCode. Instead you test values explicitly in the same program line as you may perform the branch.

Sometimes you can eliminate branches by using logic instead - for instance you can get the absolute value of a number two ways - by negating it on condition that it is negative, or by extracting the sign bit, adding it and exclusive-ORing its bit value with every bit of the result. This has the effect of conditional twos-complement negation, by complementing and incrementing the value if the sign bit is set, and leaving it alone if it's positive. This in-line code may be more effective than a conditional test.

Another way to eliminate branches is to merge them. Rather than sift through possible values one at a time, with conditional branches for each possible case or pattern, munge the values into a small range and index that into a table to find the address of the corresponding code. This gives one jump that cannot be predicted, instead of many where the redirection might go awry. If you must sift values individually, put the most likely case at the top of the list. In some cases this can double the speed of a critical routine.

The only way to be sure whether or not these techniques will help you is to try them in a particular case - but bear in mind that VPcode will perform differently depending on the particular target processor. In rare cases when top performance is essential, you might include several versions and chose them on the basis of an interactive test.

But get it working first, before you try to optimise the code. Many potentially perfect programs have never been finished, while pragmatic ones are tested and working for a living.

Further reading

This chapter is an overview of VP, specially written to introduce Amiga developers to this core concept in the new system. The next couple of pages are desnsely packed with reference material, in the style of the programmers reference cards supplied with classic silicon processors.

There's plenty more information about VP later in this volume, especially in the chapter about Developing Tools. The *VP Reference* and *VP macros* PDF files on your Amiga development CD contain almost two hundred more pages of useful information for the VP programmer.

VP Programmers' Quick Reference

Instruction Types Description

als c allocate structure on stack
 bc cond conditional branch
 bcn cond branch conditional (not likely)
 bcp cond branch conditional (probable)
 cpbb src,dst,len p copy block of bytes
 cpbi src,dst,len p copy block of integers
 cpsb src,dst,tmp p copy string until nul, autoincr.
 cpy src,dst copy source to destination (*all types*)
 d_i d double bit pattern to 2 integers
 ent subroutine entry
 entih interrupt handler entry
 f_i f float bit pattern to integer
 go tag t goto, unconditional jump
 gos tag t gosub, subroutine call
 i_d i 2 integers bit pattern to double
 i_f i integer bit pattern to float
 i_l i 2 consecutive ints to long
 l_i l long to 2 consecutive ints
 ncall px,method named object call
 noret return which is never reached
 qcall tool, {x : x} virtual call. Params in and out
 ret return from subroutine
 schk stack check
 sync align target address (obscure)
 syncreg preserve registers (obscure)
 zap undefine register

Expression Types Description

add + ilfdpc add
 and & ilc bitwise and
 and && ilc bitwise logical Boolean and
 asr ilc arithmetic shift right
 b2i b sign extend byte to int
 bclr ic bit clear
 bset ic set bit
 c2i cond conditional to int (0..1)
 d2f d double to float
 d2ir d double to int (round)
 d2it d double to int (truncate)
 d2lr d double to long (round)
 d2lt d double to long (truncate)
 div / ilfdcdvise
 divh hc fixed point divide
 divu ilc unsigned divide
 f2d f float to double
 f2ir f float to integer (round)
 f2it f float to integer (truncate)
 i2d i integer to double
 i2f i integer to float
 i2l i sign extend int to long

i2p i integer to pointer
 imm ilfdpc expression to immediate
 l2d l long to double
 l2i l long to integer
 ld bsilfdp load from memory
 ns ni nl nd (*non-aligned load types*)
 lsl << ilc logical shift left
 lsr >> ilc logical shift right
 mul * ilfd multiply
 mulh h multiply fixed point integer
 not ~ ilc bitwise not
 not ~! ilc bitwise/logical not
 or l ilc bitwise or
 orll ilc logical bitwise Boolean or
 ord fd true if ordered
 p2i p pointer to integer
 rem % ilc remainder
 remu ilc unsigned remainder
 s2i s short to integer
 st bsilfdp store to memory
 ns ni nl nd (*non-aligned store types*)
 sub - ilfdpc subtract
 swbi reverse byte order in word
 uno fd true if unordered
 xor ^ ilc bitwise logical xor

Conditions Description

ge >= greater than or equal
 geu greater than or equal unsigned
 gt > greater than
 gtu greater than unsigned
 le <= less than or equal
 lt < less than
 eq = equal
 ne <> != not equal
 bit ? true if bit 'n' (0..31) is set
 nbit !? true if bit 'n' (0..31) is clear

Abbreviations for Types

b	Byte (8 bits)		
c	Constant	l	Long
cond	Conditional	n	Non-aligned word
d	Double	p	Pointer
f	Float	s	Short (16 bits)
h	Fixed (16.16)	t	Tag
i	Integer	u	Unsigned (32 bits)

Instruction suffixes, formats and sizes

Suffix	Name	Size	sizeof()
.b	Byte	8 bits	1
.s	Short	16 bits	2
.i	Integer	32 bits	4
.p	Pointer	32 bits	4
.f	IEEE 754 Float	32 bits	4
.l	Long	64 bits	8
.d	IEEE 754 Double	64 bits	8

VP Processor Model

VP has five banks of general purpose registers: Int, Long, Float, Double and Pointer, numbered from zero upwards.

General purpose registers

Int	32 bit integers
Long	64 bit integer registers
Float	32 bit IEEE-754 floats
Double	64 bit IEEE-754 floats
Pointer	32 bit pointers

Special purpose registers

gp	Globals pointer
lp	Link pointer
sp	Stack pointer
pp	Parameter pointer (pre-call SP)
si	Signature integer

Do not zap or modify special registers

VP has no condition code register flags or program counter; conditional branches must be immediately preceded by a compare instruction.

High-level language macros

allocstruct <size>, freestruct <size>
printf "<format string>",varargs
tracef "<format string>",varargs
ktrace "<format string>",varargs
repeat / until <condition>
for <count>,<integer register>
next <integer register>
while <condition> / endwhile
loop / endloop
break
breakif <condition>
continue
continueif <condition>
bool <condition>,<label>
if <condition>
elseif <condition>
else
endif

Assembly-time macros

```
.if <condition>
.elseif <condition>
.else
.endif
.include <file>
.align
.macro / .endm
```

Note the preceding dot to differentiate these from homonyms which are evaluated at run time.

Structures

```
structure <offset>   count := <offset> (default 0)
struct <name>,<size> name=count; count+= size
int32 <name>   name=count; count += 4
float32 <name> name=count; count += 4
pointer <name> name=count; count += 4
int64 <name>   name=count; count += 8
float64 <name> name=count; count += 8
size <name>   name = count
```

Printf & Tracef formatting

%<flags><width><"."precision><qualifier><conv>

Format Flags

- left justify
- # use alternate form
- + force signed prefix
- 0 display with leading zeroes
- space space prefix if positive

Format Qualifiers

- h 16 bit short integer
- l 64 bit long integer
- L long double (same as double)

Format Conversions

Char	Type	Varargs
%	output '%'	0
b	binary	64
c	char	64
d	Decimal	64
e,f,g,E,G	Floating point	64
i	signed integer	64
o	signed octal	64
p	pointer	64
s	string	64
u	unsigned decimal	64
x,X	Hex	64

Introduction to Java

The Java programming language is an object-oriented, architecture-neutral dynamic language. It was originally designed when C++ proved unsuitable for a project to develop advanced software for consumer electronics.

The syntax of the Java language is similar to C and C++. This means that programmers familiar with C and C++ can learn the language with minimal training. Many of the less well understood features of C++ have been omitted, primarily those concerned with operator overloading (although the Java language does support method overloading), multiple inheritance and extensive automatic coercions. Another source of complexity in C and C++ applications, storage management, is dealt with in Java technology by automatic garbage collection - the freeing of memory not being referenced.

Developers have found that the Java language's simplicity, combined with its powerful object-oriented features, make it a productive language for developing applications, and its architecture neutrality, implemented by the Java Virtual Machine, has made it the ideal language for the World Wide Web and Internet.

Hello World

Here is a Java language version of the canonical demonstration program 'Hello World':

```
package demo.example.j

public class Hello
{
    public static void main (String args[])
    {
        System.out.println("Hello from Java");
    }
}
```

The resulting output will be the words 'Hello from Java' printed to the system's standard output device.

This program declares a class called Hello and a method called main in the package `demo.example.j`. As the Java language does not support stray variables or global functions, the skeleton of every application must be a class

definition. When a Java application is run it is necessary to specify the class to be run. The interpreter then invokes the `main()` method defined within that class. This method controls the flow of the program, allocates whatever resources are needed, and runs any other methods necessary. The package definition allows classes within the same package to access one another's members.

Bear in mind that this `System.out` is not one hundred per cent pure Java technology, as not all Java platforms have the concept of standard input/output streams. The alternative would be to use a Graphical User Interface.

The Java Platform

The Java platform makes it possible to run Java applications on many different types of machines with minimal rewriting of code. The portability of the Java language is achieved by two mechanisms: the translation into Java bytecode and the implementation of the Java Virtual Machine, as well as some aspects of the language itself.

Java bytecode

Java technology was designed to support applications on networks. Since networks are often composed of a variety of systems with a mixture of processor and operating system architectures, the Java language compiler generates an architecture-neutral object file format.

The compiler does this by converting Java language source (`.java`) files into bytecode instructions which are easy to interpret on any machine, and easily translated into native machine code on the fly. These bytecodes are then placed into class (`.class`) files. One class file is generated for each class in the source.

The Java Virtual Machine

The Java Virtual Machine is an abstract computer which runs programs compiled into Java bytecode. It is 'virtual' because it is generally implemented in software on top of a real hardware platform and operating system. As all Java programs are compiled for the Java Virtual Machine, programs will only run on a platform if the Java Virtual Machine has been implemented for that platform.

The 'virtual hardware' of the Java Virtual Machine can be divided into four basic parts - the registers, the stack, the garbage-collected heap and the method area. These parts must exist in every Java Virtual Machine implementation.

The Registers

The Java Virtual Machine has a program counter and three registers that manage the stack. It has few registers because the bytecode instructions operate primarily on stacked values. This stack-oriented design helps keep the Java Virtual Machine's instruction set and implementation small.

The Java program counter (or pc register) keeps track of the next instruction to be executed. The other three registers (optop register, frame register and vars register) point to various parts of the stack frame of the currently executing method.

The Method Area

The bytecodes for a program are contained in the method area. The program counter always points to some byte in the method area, and following the execution of an instruction is set to the address of the instruction which follows the previous one.

The Stack

On the Java platform, the stack is used to keep the state of each method invocation, to store the parameters for and results of bytecode instructions, and to return values from methods. The stack frame of a currently executing method keeps track of the state of that method invocation using three sections: the local variables section, the execution environment section and the operand stack. These are pointed to by the vars, frame and optop registers respectively.

It should be noted that the optop register points to the top of the operand stack, and that as this is the uppermost stack section the optop register therefore points to the top of the entire Java platform stack.

The Garbage-Collected Heap

While the stack is concerned with methods, the heap contains the objects on which the methods act. When memory for an object is allocated with the new operator, it comes from the heap. Although in the Java language memory may not be freed explicitly, the runtime environment automatically frees those objects which are no longer being referenced. This process is known as garbage collection and is explained more fully later in this chapter.

The Libraries

Several libraries of utility classes and methods are available in the complete Java system. These libraries include the basic Java language classes, the Input/Output package, the Java language utilities package and the Abstract Window Toolkit. These core libraries are implemented using our compact and architecture-neutral VP code, and are smaller than the conventional Java technology implementation, which uses its large class libraries mindful of neither size nor speed.

The Java Language package contains the collection of base types that are always imported into any given compilation unit. This includes the declarations of Object and Throwable, the base classes for objects and exceptions, as well as threads and 'wrapper' classes for the primitive data types. The Input/Output package contains the rough equivalent of the Standard I/O Library found on most Unix systems. Classes in the Utility Package include common storage classes and special use classes such as Date.

Transferring Java applications easily from one window system to another is enabled by the Abstract Window Toolkit, which contains classes for basic interface components such as colours, fonts, windows and panels.

Object-Oriented Programming

An object is an encapsulated piece of code whose state can be manipulated by methods specific to the class to which it belongs. An object is an instance of the class to which it belongs.

The variables belonging to the instance are encapsulated within the object. An object's methods are the only means by which other objects can access or alter its instance variables, although classes may declare their instance variables to be directly accessible by other objects. This is discussed later, under the heading Access Control.

Classes

A class is a software construct that defines the data (state) and methods (behaviour) of the specific objects constructed from that class. Objects are obtained by instantiating a previously defined class, and many objects may be instantiated from one class definition. Any other object may create an object by declaring a variable to refer to it and then allocating an instance of the object.

Subclasses

A subclass is a mechanism which allows new objects which are similar to already-defined objects to be defined in terms of those existing objects. A subclass inherits the methods of a single parent class, and contains additional methods of its own. It may have subclasses of its own. Methods in the subclass override those in the parent class. The code for a subclass includes the name of its parent or *base class*. All classes in Java technology ultimately inherit from the class `Object`.

Single Inheritance

A subclass may inherit from only one parent class. In C++ a subclass could inherit from multiple parent classes. This could cause problems in cases where variables or methods in the parent classes had the same names. Whereas single inheritance simplifies things, it also causes the problem that where methods in another class might be useful to a subclass, they cannot be inherited. Java technology solves this by use of software interfaces.

An interface declares methods, but does not include instance variables or implementation code. As the implementation code is contained within the class implementing the interface, the implementation may be altered without affecting the interface itself.

Thus classes may implement as many interfaces as are needed, but may inherit from only one parent class. One drawback to this is that a class must implement all methods of a given interface, a requirement which often leads to empty methods and code bloat.

The inability to inherit the implementation of an inherited class means that less code is re-used than in a multiple inheritance model. This problem is solved in Amiga technology by the use of tools rather than classes and methods.

Class Variables and Class Methods

The Java language follows the conventions from other object-oriented languages in providing class methods and class variables. Unlike instance variables, of which there is one in each separate object created from the class, the class variable is local to the class itself and the single copy is shared by every object instantiated from the class.

Class methods are used when a particular behaviour is common to every member of a class, especially if knowledge is required which can only be obtained from other instances of the class.

Access Control

When declaring a new class it is possible to set the level of access permitted to its instance variables and methods.

The access level may be public, protected, package or private. These indicate whether subclasses of the class, classes in the same package of the class, and any other classes may access the member variable. The default is package, which allows members of the same package to access members.

Packages

Packages are collections of classes and interfaces which are related to each other in some useful way. They are created by storing the source files for the classes and interfaces of each package in a separate directory in the file system.

The primary benefit of packages is that they allow many class definitions to be organised into a single unit. The secondary benefit is that the useful class members are available to the classes within the package which might need them, but not to classes defined outside the package.

Dynamic Aspects of Java Technology

The Java language was designed to adapt to changing environments. Classes are linked in as required, and incoming code is verified before being interpreted. Since Java is an interpreted language, more compile-time information is available at runtime.

JIT, the Just In Time Compiler

Despite the advantages of run-time interpretation, it has the drawback that it can be very slow. The JIT compiler gets around this problem by compiling Java bytecodes to native machine code at runtime. Thus execution of a Java class invokes the JIT compiler built into the interpreter, which compiles the bytecodes to native instructions. A JIT compiler is usually optimised towards a certain platform.

Amiga's Java platform does not require a JIT compiler as such; this functionality is inherent in our standard translation mechanisms.

Avoiding Constant Recompilation

The 'fragile superclass' or 'constant recompilation' problem occurs in C++ as a side-effect of the way that object-oriented features are normally implemented. Any time a new member is added to a class, those classes which reference that class may break as a result of the change.

The Java language solves the constant recompilation problem in several stages. The major step is in Java language compilation; references are passed to the interpreter as symbolic references, rather than being converted into numeric values. It is the Java language interpreter which performs final name resolution and determines the storage layout of objects. Conventional Java technology interprets bytecodes on the fly so changes to the source do not require recompilation of all affected classes. A similar approach to translation solves this problem in the Amiga system.

Garbage Collection

Garbage collection is central to Java programming. It achieves high performance by taking advantage of the natural pauses in user behaviour. During these idle periods the Java run-time system automatically runs the garbage collector in a low-priority thread. Unused memory is gathered and compacted, dynamically freeing memory resources.

The Java memory management model is based on objects and references to objects. All references to allocated storage are through symbolic *handles*. The Java memory manager keeps track of references to objects, and garbage collection frees the memory used by objects which are no longer being referenced.

The dynamic and automatic nature of memory management in Java technology eliminates the need for explicit memory management, which in C and C++ has proved a major source of bugs, memory leaks and poor performance. Within our Java implementation, garbage collection can be interrupted by system events.

Java and Embedded Systems

It is critical that embedded or real-time systems are designed to cope with the timing constraints imposed by the world outside the computer - random, short-lived external signals must be responded to before the data contained within them is lost. Such systems normally have few resources and require an operating system with a small footprint. Sun's EmbeddedJava technology has shrunk the large footprint of Java technology, and the Java language has advantages over the C currently used in most embedded programs.

While C has permissive operations which can lead to undisciplined coding practice and unstable execution, the Java language requires explicit declarations and tight coding. Also, the universal standards provided by the Java language for multithreading and shared data protection make the program easy to transfer to alternate platforms. The behaviour of our Java is identical across platforms.

Real-Time Behaviour

The translation between Java language source and Java bytecode gives Java technology platform independence, but at the cost of performance. Run-time interpretation can be slow, whereas often embedded systems require fast, known response times. The automatic garbage collector saves programmers effort and eliminates a major source of bugs, but its tendency to impose long delays at unpredictable intervals interferes with the ability to comply with real-time constraints. Our garbage collector is interruptible, which alleviates these pressures.

Our Java technology translates rather than interprets the Java language, and does so at load time rather than while the code runs, for better runtime performance.

Memory

Java technology was first developed in the desktop applications market, which has very different economies from the embedded market. We need to address both. Current implementations of the Java technology are rarely space-efficient. This is much less of a problem in the Amiga system, as the use of individual tools rather than classes offers much greater memory efficiency.

The Amiga Java Platform

Amiga provides developers with the opportunity to create devices with attractive user interfaces, complex applications and Internet access tools while minimising usage of system resources and getting their products to market faster than ever.

Amiga is a content platform which combines multimedia tools, gathered in the multimedia toolkit, and the engines necessary to run the content, such as Amiga Java.

The Amiga platform achieves portability thanks to its virtual processor layer. The Virtual Processor defines a language, VP, and a platform isolation interface which allows a minimal set of platform-dependent features to be isolated from the rest of the system. The basic translation mechanism is to load VP bytecodes and translate these into native code. Normally this process takes place when a tool is loaded, but it can also be done when the system is built.

Amiga runs Java applications using a full implementation of the Java Virtual Machine. It translates Java bytecodes into VP bytecodes, so it can work with any development environment that produces Java classes in the standard bytecode format. Unlike conventional Java technology, the Amiga Java engine was developed with the embedded market in mind, making small footprint and speed its priorities. Even so, we have taken care to ensure full compatibility.

Small Footprint

Rather than classes and objects our Java implementation relies upon thousands of compact tools which are loaded and bound upon demand. This means that Java technology can now be implemented even in the most memory-constrained environments.

Core class libraries conforming to the PersonalJava technology standard Application Program Interface are implemented in our portable VP code, which is highly memory efficient and executes quickly compared with rival systems.

Object-Based Programming

Object-based programming differs from object oriented programming in that its fundamental unit is the individual tool. Amiga tools are functions; small sections of executable code. Whereas object-oriented programming requires that for a particular method to be used the entire class must be loaded, in object-based

programming each method is an individual tool, which may be called by any object. Tools can be used to build classes and objects, and thus program in an object-oriented manner.

Amiga's Java engine implements Java language methods as individual tools. As in the Java technology, there is no fragile superclass problem. **Binding** - replacing the tool's name with a pointer to the tool's code - occurs after translation, much as the Java language interpreter, rather than the compiler, performs final name resolution and determines the storage layout of objects.

Virtual Processor code, in which Amiga tools are written, occupies a middle ground between the expressive but space-consuming objects used in object oriented programming, and low level, hard graft, assembler language.

Amiga Dynamics

As in Java technology, Amiga tool loading is dynamic; tools are loaded when required then remain in memory until no longer referenced. Any tool not being referenced may be flushed from memory if the memory is needed.

The Jcode garbage collector scans dynamic memory areas for objects and marks those which are no longer referenced. These allocations are then freed. The garbage collector is configured to run on request, or whenever a set amount of memory has been allocated following the last garbage collection, as well as when the working set of allocated memory reaches a high water mark.

Real-Time Multithreading

Amiga has been designed to support real-time behaviour. Message passing between tools can be either synchronous or asynchronous depending upon task requirements, and various other objects of this kind, such as mutexes, signals and semaphores, are supported.

Amiga supports multi-tasking and permits the simultaneous implementation of a range of scheduling policies.

In an environment where no underlying threads are used, it is usually necessary to build a thread model into the Java Virtual Machine. We directly map Java language threads onto our lightweight process model, ensuring that thread scheduling can be tuned according to the particular platform on which the threads are to be executed.

Using Amiga Java Technology

Translating a Java Language Class

The Amiga system can work with any standard Java technology development environment. It includes *javac*, which compiles Java source code into bytecodes.

The following command loads and if necessary translates a Java language class:

```
jcode <path>
```

The shell command *java* can also be used to load and translate Java.

This is a dynamic translation, so we recommend that classes translated in the fashion be kept small. *jcode* will always use a statically translated tool in preference to translating dynamically. When using *jcode* the *.class* suffix should be omitted.

The example 'Hello World' program at the start of this chapter would be executed like this:

```
jcode demo.example.j.Hello
```

Translating into Tools

Java language `.class` files can be statically translated with the `translate` command. This will produce a number of tools contained in a directory of the same name as the original `.class` file. The whereabouts of this directory will depend on the package declaration in the original Java language source. For example:

```
translate -tjcode demo/example/j/ Hello.class
```

will produce a number of separate tools in the directory `demo/example/j/Hello`.

The `-t` option specifies the translator, in this case `jcode`. Note that `-tjcode` and `-t jcode` are functionally equivalent, and that the `.class` suffix is necessary in a static translation. The tools produced can then be run using the command:

```
jcode <directory/package>
```

In this case:

```
jcode demo/example/j/Hello/
```

Running Non-Text Files

`jcode` is only used to launch text based applications from the shell. Any code which requires the Abstract Window Toolkit must be launched after that Toolkit has been started. This is accomplished by running a system image that can either launch the shell or its graphical equivalent, **eterm**. The Abstract Windowing Toolkit itself uses the Amiga multimedia toolkit to render graphics.

Amiga Java Architecture

The Amiga system enables compelling interactive content, compatible with everything from low end devices to desktop workstations. It incorporates the Amiga multimedia toolkit and the engines necessary to run the content, such as Amiga Java.

Amiga Java is a implementation in Virtual Processor Code of the Personal Java Application Program Interface class libraries, combined with a Java Virtual Machine and runtime environment. It also includes a translator to convert Java bytecode into Virtual Processor bytecode.

Our Sun-authorised implementation allows the class libraries to be implemented in a memory efficient manner, thereby allowing Amiga Java to function in embedded environments where size and performance issues would normally prove prohibitive. The advantage of translation is that a translated system works at near to one hundred per cent efficiency compared with only a few per cent for a interpreted system, as commonly used by Java Virtual Machines.

For embedded systems all translation can be carried out at system build time, rather than at load or run-time. Run-time Java technology facilities such as garbage collection and exception handling are also included as part of Amiga Java.

Java Technology and Amiga

The translation of Java bytecode is entirely transparent as far as the programmer need be concerned; the functionality of the standard library functions conforms with the PersonalJava Application Program Interface. Bookshelves are groaning under the weight of Java-related tomes. We have no intention of duplicating their content here, in so far as it is not unique to the Amiga system.

However, the re-implementation of the Java bytecode as VP code allows for significantly greater speed and minimal size. The following section of this document describes these alterations, but for more detailed information concerning the operation of Amiga's technology you should consult other parts of this book in conjunction with the *VP Reference Manual* and *Realtime Kernel Reference Manual*, supplied as PDF files on your Developer CD.

The Java Platform

The Java Platform comprises three components - the language, the virtual machine and the libraries.

The Java Virtual Machine

The Java Virtual Machine is a description of a software machine that must conform to the standard specification produced by Sun Microsystems. The Virtual machine is responsible for taking Java bytecode and transforming and executing that code on a specific processor platform.

Amiga Java implements the Standard Java Virtual Machine. However, the techniques and details of the implementation vary greatly from a standard Java Virtual Machine implemented using Just In Time 'JIT' compilation.

The Java Libraries

The Java libraries provide the Java environment with all of the services available on the particular host platform. By standardising the libraries and using the virtual machine, the dependency on the underlying platform is greatly reduced from a developer's perspective.

The libraries' standard application program interfaces make it possible to write code that is binary code compatible across a number of platforms. This benefit is used to great effect on the World Wide Web, where Java class files can be downloaded to hosts such as mobile phones and set top boxes among many others, and executed on the virtual machine, extending its functionality in an almost unlimited way.

The Java technology defines a set of core libraries which form a part of the standard Java development and execution environment, and a set of standard extension libraries that may be used for specific areas of application, or on specific platforms or environments.

Virtual Machine and Library Testing

To ensure predictable and consistent behaviour between Amiga Java, including both our implementation of the Java Virtual Machine and the Java Libraries, and other platforms, a complete suite of test libraries has been produced.

To ensure that each of our libraries implements the Java technology specification, a test suite of 2,500 individual test methods has been produced by Amiga in association with Plum Hall, the coding and testing experts. More importantly, Amiga has been given access to Sun's TCK test suite, thereby ensuring the compliance of Amiga Java to Sun's own specification.

Amiga Java's implementation of the Java Virtual Machine specification is also tested using the 3,000 separate test cases developed by Plum Hall, to ensure complete compliance with the standard.

All of these tests are conducted externally from the Java team, to ensure objective evaluation against the Java technology specification. This rigorous, independent and objective testing effort, managed by Plum Hall, guarantees full standards compliance for all of the Amiga technology that executes Java bytecode.

Transporting Amiga, Java Technology Edition

Most execution environments for the Java language are written for a specific machine type. For example the implementation of the maths libraries will be optimised for a specific host machine, and the back end of any Just In Time compiler will be specific to a processor or chip set.

Almost without exception the whole of Amiga has been written in the Virtual Processor code, machine-neutral format used extensively by our operating system and the Amiga content foundation. This means that the entirety of Amiga Java can be moved from processor to processor, and from system to system with minimal overhead.

Java is designed to be portable, and our compliance with Sun's standardisation efforts boosts that potential, because our Java system has passed the same tests as other validated systems. Even beyond that, the generic, portable core of the Amiga Java implementation ensures great compatibility between Amiga systems, from the biggest box to the near-invisible appliance. You can maximise the market for your programs by designing them from the outset to match this all-encompassing potential of the new Amiga system.

Objects and Amiga

Amiga is composed of thousands of tiny, modular tools. Each is a thread of executable code, akin to a Java method or function, that can be dynamically loaded into a runtime environment.

The footprint, in terms of both static ROM or Flash memory and dynamic RAM memory, is exceptionally small, at around two megabytes. This is enabled both by the very small footprint of each of the tools, and the fine grained dynamic sharing of these tools at run time. Tools are fully re-locatable and re-entrant which means that they can be shared across multiple processes. As a consequence of this only one copy of any tool ever needs to be stored in memory.

When a class is designed in Amiga's VP code, the code for the associated methods may be placed within the class itself. However, it is also possible to remove the code for each method and then place it into a separate tool. This can then be called from memory, should it be needed. The method left within the class, in these circumstances, is little more than a call to the relevant tool. By exiling method code to independent tools which will only be summoned if needed, the amount of memory initially needed to load the class is scaled down dramatically.

Tools are not objects in the usual Object Orientated Programming sense. Like objects they are modules designed for easy manipulation by the programmer, but tools are not used as receptacles for data. Since they are functions and contain executable code, tools are more aptly compared to methods. In the same way that methods can perform operations upon the data contained within objects, tools can be used to act upon resources of data held within data structures. The principal difference between a method and a tool is that a tool exists as a separate unit rather than as a subset of a class.

Amiga's Class Tools support inheritance, including calling up to parent classes to find methods by signature matching at run-time. All object based behaviour is supported by Amiga Java. This means that there is very little semantic gap between the Java bytecode, and the underlying architecture of Amiga Java. We have all that Java can offer, and more besides.

Multithreading

Applications that are multithreaded can do more than one thing at a time. They run as separate processes, so that when the user selects one operation the application continues to respond as before and can , Multithreading is a hallmark of the new Amiga, and vital to the ease of use of applications.

Consumers do not expect one appliance to stop working because they're using another, yet the 'appliances' simulated by desktop computers often work that way. Even some Classic Amiga applications suffer from a lack of threading. Programs that do not make proper use of threads will be swept aside by others that do, because the rivals will be easier and more intuitive in use.

Java technology has threading built into the core of the language. In fact, threading is an underlying assumption of the Java object model. The Java class `Object` has threading methods defined in its interface. This means that any Java operating environment must implement a thread or lightweight process model of some kind.

Because it provides built-in support for lightweight processes at its core, Amiga Java can directly map Java threads onto the lightweight process model. Thread scheduling can be tuned on a per-platform basis to achieve high reliability and performance.

Normally each thread in the process group shares process data structures like the signal table, memory object and environment list with the other process in the group.

Within the Amiga implementation each Jcode thread for a virtual machine has the same memory objects, environment list, signal handlers and statics. As each Jcode thread is a separate process they can consequently be assigned priorities individually.

Using Amiga Java

The Development Environment

Amiga Java operates at the bytecode level, so it has the ability to work with any standard Java development environment that produces Java classes in the standard bytecode format. Amiga Java fully includes `javac`. This may be used to compile Java source code into bytecode form.

The interface is identical to that when using `javac` under Sun's Java Development Kit. At the shell prompt you type:

```
javac <source>
```

Thus, to compile the source file `FRED` to Java bytecodes, you'd type:

```
javac fred.java
```

Jcode Translation and Execution

Java technology mechanisms such as the Remote Method Invocation mechanism, RMI, Java Beans, the Java plug and play component technology, and the underlying reflection mechanism are inherently dynamic features of the Java library and runtime system. To make these technologies viable the underlying operating environment must also be dynamic.

Amiga Java supports dynamic class and method loading at its core. It uses dynamic tool loading, among other features, to implement the Java run time environment, so there is no conceptual gap between Amiga Java's functions and facilities and the dynamic environment which Java technology demands.

Legacy operating systems tend to be forced to implement many features that are not native or inherent in the underlying environment in order to produce a realistic Java Engine. This has impact upon not only memory footprint but also performance, as the semantics of the implementation twist and turn through legacy concepts and features. We have no such baggage to weigh us down.

Mixing Dynamic and Static Execution

Amiga makes no assumptions about the way in which individual platforms will take advantage of its features. It is possible to "brew" specific Java execution

environments for each platform on which Amiga Java is implemented. This often amounts to mixing dynamic and static components of an environment to match specific needs.

The following section describes how Java bytecode can be dynamically executed in Amiga Java and how we handle the ability to "blend" in Java classes at build time if required.

Dynamic Run-Time Execution of Java bytecode

Any proper implementation of the Java environment must allow Java Classes written in Java bytecode to be dynamically loaded by ClassLoaders specified in the definition of the Java Virtual Machine. Amiga Java achieves this by making novel use of Amiga's unique technology.

The following examples assume a test class is available called `andi.class`. It is contained within the file `Java jvmsup.zip`, which is itself usually contained within the file `j.zip`. Once unzipped the `andi.class` file is found in the directory `j/tst/jvmsup`. When running Java classes always work from the root directory.

The `jcode` command is used to load and if necessary translate a Java class, so for example to run the `andi.class` example class, you would type the following from the root:

```
jcode java/tst/jvmsup/andi
```

Note that the full path is typed and the extension of the class file is not specified.

Running The Application

The method described next is used as an entry point to the class specified in the `jcode` command:

```
public static void main (String [ ] args)
```

The name given to the `jcode` commands needs to be the absolute name of the file. Note that this is a dynamic translation, and so we recommend that the size of the classes be minimised.

Static Build-Time Translation of Java bytecode

Some uses of Java technology will include embedded applications wherein all of the Java classes required will be known when the application or device is manufactured. In this case it is possible to "burn in" only the required classes for the particular device's application. The ramification of this topic are beyond the scope of this manual. For now, you simply need to know is that it is possible.

Translating Programs with Multiple Classes

It is unnecessary to create a new directory for each class as the jcode translator will do this automatically when it translates the classes.

Directory Mappings

For the following example class, the standard directory mappings for code locations imply that:

```
java/tst/jvmsup/andi.class
```

should appear in the directory marked:

```
< root>java/tst/jvmsup
```

Garbage Collection

The Jcode garbage collector uses a mark-sweep algorithm to detect allocations which are no longer being referenced and then frees them. In other words, it scans dynamic memory areas for objects and marks those that are referenced.

The garbage collector can be configured to run whenever a set amount of memory has been allocated following the last garbage collection, as well as when the working set of the Jcode process reaches a high water mark.

The garbage collector will allocate memory for both object and buffers of data to be used internally. Each allocation can be flagged to be searched for pointers by the garbage collector during the mark phase. Byte and char buffers will never contain pointers to other memory allocations and therefore do not need to be searched. Don't be tempted to pack pointers into such areas - if you try this, your code will soon fall over.

The garbage collector mark phase starts with the contents of the registers and stacks of each Jcode thread, and searching for pointers to allocated memory. Each allocation found is then recursively searched for pointers to other allocations until a maximal set is found. Any allocations not found by this phase are marked and freed by the sweep phase.

An allocation is considered 'live' if there is a pointer to it held in another live allocation, the stack of a Jcode thread or a register of a Jcode thread. The pointer may be to anywhere in the interior of the allocation or to the immediately preceding word.

The Amiga applet viewer

An applet viewer is a program designed to run applets outside a browser. Amiga's implementation of Java technology requires the use of such a viewer. Some other systems also make use of an applet viewer, but the Amiga one has unique advantages. In the first place, it uses multiple Java Virtual Machines. It also benefits from all the usual advantages of Amiga's technology, such as reduced size and increased speed.

The Java Development Kit specification defines a command-line application which can be used as an applet viewer. When provided with an html file, this viewer is designed to identify the tags that define an applet, and tends to disregard irregularities in the rest of the html source. The Amiga applet viewer behaves in a similar fashion.

In order to use the Amiga applet viewer, an html file must be supplied. This file contains references to a Java class, and related resources on a remote host. These must be loaded across the network for the applet to be run. The special class loader provided for this purpose is described below.

The html file is parsed. Where the source of the html file defines an applet, an instance of `ElateAppletStub` is created. These stubs are generated in accordance with the values of the tags in the source.

Each applet is run within a special environment called `AppletStub`. An applet stub is an interface which is used as a container for a particular applet. All stubs of this sort are contained within an application called the `AppletContext`. If more than one applet are specified by the html file, these applets will all run within the same context.

The Applet Viewer is written in the Java language and thus requires the use of the jcode harness. In order to parse an html file using the applet viewer, the following command should be entered, followed by the URL or name of the html file:

```
jcode com/Elate_group/applet/ElateAppletViewer
```

Only the first file listed as an argument will be parsed.

The class loader

A class loader is supplied so that the necessary classes and resources may be loaded from a remote system. These files may either be loaded directly across the network, or can be placed in an archive, which is effectively a 'zip' or 'jar' file. A description of these jar files is given in the next section. The class `HttpClassLoader` is contained in the directory `sys/j/applet`.

Java archives

These **jar** files are sometimes used to store files that are being loaded across a network. The advantage of this approach as opposed to directly loading from the remote host is the reduction in the number of network connections. A jar file of this sort is effectively a zip file without compression, which contains some security features such as signatures. In the current version of Amiga Java, signatures are disregarded, and the archive is generally treated like a zip file.

An archive of this sort may be created this way:

```
zip -0r archive.jar *.class
```

The above command will create the jar file `archive.jar` as a zip file with zero compression. All class files from the current directory and from all its subdirectories will be placed into the archive.

Java Beans

The Java programming language can be used to write reusable, self-contained software components known as **Beans**. As described overleaf, Beans as supported by Amiga Java behave in a fashion that conforms to the standards set in Sun's Java Development Kit 1.1.7 specification. The use of the self-contained modules of Java Beans highlights the similarity in design philosophy that helps to make Amiga and Java natural partners.

Java Beans are created using the `JavaBeans` application program interface. Visual application builder tools can then be used to compose these Beans into composite components, applets, applications and servlets. A **servlet** is a small program designed to supply extra levels of functionality to servers that support Java technology.

The features of each Bean include properties, public methods and events. Properties define aspects of a Bean's behaviour and appearance which can be altered during design time.

The features of a Bean can be examined by a JavaBeans-enabled builder tool using a process known as **introspection**. Bean feature names follow specific rules, known as design patterns. The builder tool is thus able to examine the patterns and determine the features of the Bean. The `java.beans.Introspector` class is responsible for examining the patterns and identifying the features.

The property, method and event information associated with a Bean are stored with a corresponding Bean Information class. This class implements the `BeanInfo` interface, and explicitly lists those features which are to be made visible to the builder tool.

After identifying the features of a Bean, the builder tool exposes them for visual manipulation, allowing them to be customised at design time. This modification can be performed either through the use of a property editor or by using a more sophisticated Bean customiser.

Beans are kept by the builder tools in a palette or toolbox from which they may be selected, modified, or combined with other Beans as required. Several Beans may be combined to form an applet, an application or a new Bean. A Bean's properties may be altered and the new state stored, so that it can be restored at a later date complete with the new modifications. This attribute is known as **persistence**.

Communication between Beans takes place through the medium of events. Different Beans may be capable of sending or receiving diverse events. The builder tool identifies which events a Bean is capable of receiving, and which it is able to send.

Java Bean methods are the same as ordinary Java methods. They may be called from other Beans, or from a scripting environment. As a default, all public methods are exported.

Unicode

Java uses 16 bit Unicode characters, eliminating the cultural imperialism of 7 bit ASCII and 8 bit ANSI codes, in which programmers outside the USA might be unable to represent the letters of their own name! Unicode has the ASCII printing characters in their usual places, from code 32 to 126, and dozens of other alphabets in codes up to 8191. The next 4096 codes are reserved for symbols and punctuation, including arrows and geometric shapes. The biggest group of codes is reserved for 'Ideographs' - the Han characters used in China, Japan, Korea, Taiwan and Vietnam.

There's substantial space reserved for expansion. The potential disadvantage of Unicode is that English text might take twice as much space as would otherwise be required. This is addressed by a derived standard, UTF-8.

UTF-8 is an important variant of Unicode, which supports the full 16 bit code packed into one, two or three bytes, so that the standard ASCII codes can use single bytes, with two bytes for codes up to 2047, which include Hebrew, Cyrillic, Arabic and basic Greek.

Three bytes are needed for the rest, such as dingbats, technical symbols and eastern alphabets like Tibetan, Tamil, Thai and Taiwanese.

This table indicates how seven, 11 and 16 bit codes are packed into one, two or three bytes by UTF-8.

Start	End	Bits	Binary	Byte	Sequence
\$0000	\$007F	7	%0xxxxxxx		
\$0080	\$07FF	11	%110xxxxx	%10xxxxxx	
\$0800	\$FFFF	16	%1110xxxx	%10xxxxxx	%10xxxxxx

Web connections

Java is an emerging standard, well documented on the Internet. The definitive Java site is at <http://java.sun.com>. Usenet discussion of Java centres around the `comp.lang.java` newsgroup. Gamelan maintains a large directory of Java resources and programs at http://www.io.org.mentor/J__notes.html. For complete information about Unicode, visit <http://unicode.org/>

The Amiga C Compiler

The Amiga Native C compiler is composed of several programs, each of which is responsible for one stage in the compilation process. This section describes how these components work together to translate C code into Amiga executables.

vpcc	Front end that invokes programs to execute other compilation stages
vpcpp	C pre-processor
vpcc1	C to VP2 code translator
vpas	Assembles compiler assembler output to object files
vp1d	Links object files into an Amiga assembler file
asm	Amiga assembler

Front End

To compile C source code, the Amiga executable **vpcc** is invoked. This program is referred to as a front end because it co-ordinates and simplifies execution of the series of stages that make up the compilation. **vpcc** understands a set of command line arguments, listed later in this chapter, and translates them, along with the names of C source files, into a series of invocations of the other components of the compiler.

Each file given on the command line must have a recognised suffix. For C this will be `.c`, and for C++ `.cxx`, `.cc`, `.cpp` or `.C`

Pre-processor

The pre-processor, **vpcpp**, interprets and acts upon all pre-processor directives in a given C source file. Pre-processor directives are those lines of a C program that begin with the hash (`#`) symbol. The output of the pre-processor is also a C source file, but one that lacks any pre-processor directives. Loosely speaking, the effect of running the pre-processor is to replace defined symbols with their values and to handle conditionally included sections of code. The pre-processor is a port of the standard GNU pre-processor.

C to Assembler Translator

vpcc1 converts a single C source file into a VP2 assembler file. This program assumes that if the C source file uses pre-processor directives or defined symbols, it has already been passed through the pre-processor.

The translator is a customised version of the standard GNU C translator, re-targeted to generate Amiga VP2 code and ported to run as an Amiga program.

Object Files Generator

vpas processes the .s assembly file output by the compiler to produce an object file with the suffix .o, suitable for the linker. This object file contains VP assembly instructions rather than binary code. Bear in mind that that vpas does not assemble VP or native code .asm files in the same way that asm does. It does another type of 'assembly', specifically designed to suit the compiler's output.

Linker

The Amiga system relies on dynamic binding of procedures and does not provide for inter-tool binding of data (sharing of static or global data between tools). The model of linking used by the compiler is quite different from the conventional one.

The linker operates on the .o object files output by vpas. The result of linking together a set of assembler files is that a single assembler file is processed by the Amiga assembler, typically to create a complete program that is a single entry point tool. Global and static data for the program are then stored in the Per Process Data Block (Per Process Data Block), a dynamically allocated block of memory. The exception is when an Amiga tool is being generated which can then be called from within a VP2 coded program or application. Tools generated are treated the same as any other Amiga tool and can be dynamically bound.

In addition to the compiled code, vpld generates prologue code, which does various kinds of initialisations when the program begins execution, and epilogue code that performs clean-up operations before the program terminates. By default, it also ensures that library support functions and definitions are included. A list of the command line options understood by vpld appears later in this chapter.

The linker also:

- Determines the total global data storage requirements of the program
- Establishes a mapping from global variable names to their offset in a global data storage area
- Maps variable names in accordance with their scope. A variable's scope might be global to the entire program or local to a particular file.

- Resolves all references to global data items such that they refer to the correct location in the global data storage area.
- Emits an initialisation routine that allocates the global data storage region and initialises it in accordance with the supplied initial values of the global variables.

The linker makes two passes over each input file. As each file is processed, the linker interprets the pseudo-operations and constructs a memory map of the Per Process Data Block. At the end of its output file the linker emits a series of symbolic equates that define the mappings of global and static data into the Per Process Data Block. The symbols defined in this section are used within instructions that reference global data. Consider the following snippet of C:

```
int bar;

void foo(void)
{
    bar = 1;
}
```

This results in the compiler generating (amongst other things) the following code:

```
extern _bar,4,4,0
...
cpy.i 1,r0
cpy.i r0,[gp+_bar]
```

The `.extern` pseudo-operation informs the linker that a global variable called `bar` has been declared and provides information about its size and alignment requirements. Notice that the variable is referred to in the generated code via the addressing mode `[gp+_bar]`. After its second pass is completed, the linker will have decided where the variable `bar` resides within the Per Process Data Block. If it decides to place `bar` at offset 8 in the Per Process Data Block, the linker would emit the following equate at the tail end of the linked assembler file:

```
_bar = 8
```

The generated code that references `bar` correctly identifies the location in the Per Process Data Block where `bar` resides. Pseudo-operations such as the `.extern` in this example can also specify initial values for variables, and these initial value specifications are used by the linker to generate a series of instructions inside the `vp_init()` routine that initialises the values in the Per Process Data Block.

The operation of the linker is greatly simplified if it is generating a single tool without a control object. As tools have no global data, none of the processing for globals is required. The linker need only output the tool node definition and a gos into the first function encountered to set up proper entry into the tool.

Labels

Because input files are concatenated together and the Amiga assembler does not support variable and label scope in the way GCC expects, labels are mapped. To make identification of labels much easier, the compiler assists by delimiting them by special characters, currently exclamation marks. The compiler also emits pseudo-operations to specify scope.

A label may have components appended to it to make it unique within a file and unique across all files. Additionally, a label that refers to an element of a structure has an encoding of its byte offset appended: `$$<offset>` or `$$m<offset>` for negative offsets.

The fact that a "\$" cannot appear in a source code symbol but is valid, except as the first character, in an assembler label is taken advantage of and used to separate components.

Labels are case sensitive in C and C++. Nevertheless, because the assembler treats labels as case insensitive, the compiler adds a variable-length component to any label that is not completely lower case. This ensures that "oXo", "oxo", and "OxO" in a C program are handled correctly., though if you use identifiers like those you probably deserve all the trouble you get!

Labels with global scope are prefixed with a string: currently the letter 'x' followed by an underscore. This keeps them from conflicting with the assembler's reserved words such as `tab`. Labels that are emitted by the linker as part of the program prologue or epilogue are prefixed with the string `x_$$` . This ensures that these labels will not clash with other kinds of labels.

Assembler

The Amiga system assembler processes the output of the linker. The result is a single tool object, with the `F_MAIN` flag set to denote an executable tool.

Compiling C

To understand how all these pieces fit together, consider this command:

```
vpcc demo/example/c/main.c demo/example/c/proc.c -o main
```

This command compiles the two C source code files, `main.c` and `proc.c`, and produces as output `main.00`. This is done in the following four steps:

Pre-processing

Vpcc runs the pre-processor (`vpcpp`) on each C file in turn, placing each result in an intermediate file. The name of the intermediate file is derived from the original source file's name. Normally, it is deleted by `vpcc` when it is no longer needed. Therefore, in this case where C files are being compiled, the pre-processed versions of the files will be stored in the temporary files `main.i` and `proc.i`.

Translation

The next phase of the compilation is the translation of the pre-processed C code into assembler source. `vpcc` runs the translator (`vpcc1`) on each of the intermediate pre-processed files. In this case, the result is two further intermediate files, `main.s` and `proc.s`, which contain Amiga assembler source. After the intermediate pre-processed files have been translated, they are deleted.

Linking

Next `vpcc` runs the linker (`vpld`), feeding it both of the intermediate files and the standard C library as arguments. The linker resolves all global data references, and produces a final assembler source file, `main.asm`. Note that the file name used here is based on the argument given to the `-o` flag on the original command line. Once the intermediate assembler source files have been linked, they are deleted.

Assembly

Finally `vpcc` runs the native Amiga assembler on `main.asm`. The result is a main tool object `main.00`. If none of these steps encounters any errors, the program can be run. Phew!

Compilation Examples

Here are some examples of command lines that invoke the Amiga C compiler:

```
vpcc demo/example/c/main.c
```

This command will cause the file `main.c` to be compiled, linked, and assembled. The resulting output will be in `demo/example/c/vpout.00`.

```
vpcc tmp/main.c
```

That command will cause the file `main.c` to be compiled, linked, and assembled. The output will be in `tmp/vpout.00`.

```
vpcc -c tmp/main.c
```

This command will cause the file `main.c` to be compiled. into corresponding assembler source in the file `tmp/main.s`.

```
vpcc tmp/main.c -o tmp/bar
```

That command will cause the file `main.c` to be compiled, linked, and assembled. The resulting output can be found in `tmp/bar.00`.

```
vpcc tmp/main.c -ptest
```

This command will cause the file `main.c` to be compiled, linked, and assembled. The output will be written to `test/vpout.00`.

Default Paths

This section summarises where the C compiler looks for files it needs.

Include path for the pre-processor:	<code>lang/cc/include</code>
Library path for the linker:	<code>lang/cc/lib</code>
Executable for the pre-processor:	<code>lang/cc/bin/vpcpp</code>
Executable for the translator:	<code>lang/cc/bin/vpcc1</code>
Executable for the linker:	<code>lang/cc/bin/vpld</code>
Executable for the assembler:	<code>asm</code>
Run time support library:	<code>lang/cc/lib/libc.ism</code>
Pre-processor output file	<code>vpout.00</code>

Compiler Configuration and Implementation Notes

This section describes aspects of the Amiga port of the GNU C compiler that may be required by users of the compiler, particularly regarding the interfacing of assembler code with the compiler's output.

Data Formats

These are the sizes that the C compiler associates with the primitive data types:

Type	Size in bits
char	8
short	16
int	32
long	64
float	32
double	64
long double	64
pointer	32

By default, variables of type char are assumed to be signed.

Register Usage

VP registers are used by the compiler as follows:

i0-i31	Integer register set
p0-p32	Pointer register set
l0-l31	Long register set
f0-f31	Float register set
d0-d31	Double register set
sp	Stack pointer
fp	Frame pointer (any available pointer register)
gp	Global data pointer

In theory compiled programs could use any number of registers, but the above list is more than the compiler can usually work out what to do with.

Function Call Conventions

The compiler adheres to the standard Amiga system calling conventions, in that it defaults to passing parameters by value. Whether a structure is passed by value or

by reference, the structure to be passed (a copy, or a pointer to the array itself) is placed on the stack to be dereferenced by the function. For `varargs/stdarg` functions, the unnamed arguments are placed on the stack.

Stack Usage

The stack is used for passing unnamed `varargs`, `stdarg` and function parameters when the parameters overflow the number of available registers, and for storing local variables when the optimiser cannot allocate a register to store the variable for the lifetime of the function.

Debugger Support

The `-g` command line flag causes the compiler to emit `stabs/dbx` format debugging pseudo-operations. These help the debugger keep track of the code during testing.

Tracing Support

The compiler can generate code to trace the entry and the exit to every function. In addition, it can generate tracing output to indicate the call and the return from every Amiga tool call. The `tracef()` tool outputs the tracing information.

The `-mtrace-funcs` command line flag causes the function tracing output to be produced and the `-mtrace-tools` flag causes tool call tracing output to be produced. The `-mtrace-all` flag enables all available kinds of tracing output.

Tool Generation Support

The compiler recognises that it is generating code for tools rather than normal C functions when it sees the `-felate-tool` command line flag. For backward compatibility, the `-ftaos-tool` command line flag is also observed. In future `-famiga-tool` could be used to similar effect.

Packed Alignment

By default, the compiler will align 64-bit long and double data types to addresses that are evenly divisible by eight. The `-mpackalign` command line flag causes the compiler to align these types to addresses that are evenly divisible by four instead. This means a slight reduction in memory overhead with no loss of performance if the program is to run on a machine with a 32 bit data bus.

Error Messages

Error messages from the compiler are saved in a file called `vpcc.log` in the same directory that the compiler is run from. If the file already exists, its previous contents are overwritten.

Troubleshooting

exit() and taos_exit()

When **exit()** is called by a program, run time support code may perform some clean-up routines before terminating. In the case of a C++ program, for example, destructors may be called. To terminate a program immediately, bypassing this clean-up, call **taos_exit()** instead.

main()

The **main()** function is called by the C runtime start up code with the assumption that it requires two parameters and returns an integer. Currently, if these two parameters are not declared for **main()**, the program will not run properly. Most include files make this declaration indirectly, but if not, it must be declared as follows:

```
int main(int argc, char **argv)
```

Device Driver Interrupt Handlers

Interrupt handlers can be coded in C or C++ in the normal way. However, the Amiga must be made aware that it is an interrupt handler; the following line must be entered directly after any include files:

```
void <name_of_interrupt_handler>  
(void*) __attribute__  
(("interrupt_handler")  
);
```


Porting existing C code

Additional debugging and programming effort may be required to port existing applications to Amiga. Much existing C source code assumes that the size of pointers, integers, and long integers are the same - usually 32 bits. This assumption is invalid in the Amiga system. It is particularly important to be careful when passing a long parameter to a function. The compiler flag `-Wconversion` can be helpful in locating these conversion problems.

Some C programs use integers and pointers in such a way that they are assumed to be practically identical. Whilst on many platforms this does not prevent a program from operating correctly, it may create problems in the Amiga environment. The Amiga has distinct register files for pointers, integers, floats and so forth; the compiler is strict about pointer parameters in pointer registers and integer (char, short, int) parameters into integer registers.

A call to a function that was not previously declared with an ANSI-style function prototype or ANSI-style function parameter list may therefore not pass pointer parameters correctly. This example illustrates the problem:

```
foo(arg)
unsigned short *arg;
{
    ...
}
...
foo(0);
```

The call to function `foo()` will pass the parameter as an integer rather than as a pointer and the value of `arg` will probably not be zero. When the compiler is able to detect these inconsistencies, it produces error messages. To avoid this problem, use an ANSI-style prototype for `foo()` or its parameter list:

```
int foo(unsigned short *arg);
```

The use of warning options, such as `-Wmissing-prototypes`, is encouraged. Beware that old C, and ANSI C, in the absence of prototypes, and in variable-length argument lists, widens certain arguments when they are passed to functions. In particular a float is promoted to a double. Mixing the ANSI-style prototype declaration:

```
extern int func(float);
```

with the old-style definition:

```
int func(x) float x;
```

will generate incorrect code since the function will expect its argument to be a double.

Data Type Conversions

Under some very rare conditions, the compiler will generate incorrect code for some data type conversions. This is believed to be a GCC bug that arises when it generates code for 64 bit long integers. The following is the only known example:

```
f()
{
    long l2;
    unsigned short us;
    unsigned long ul;
    short s2;
    ul = us = l2 = s2 = -1;
    /* ul is assigned an incorrect value. */
    return(ul);
}
```

To work around this potential problem, rephrase the assignments like this:

```
l2 = s2 = -1;
ul = us = l2;
```

The Amiga's use of 64 bit long integers might have an impact on the evaluation of certain benchmark figures. If other benchmarks make use of 32 bit integers any comparison between them and Amiga benchmarks will not be valid.

errno

The linker handles the 4 byte global variable `errno`, which is declared in `errno.h` and is part of the Amiga kernel. Programmers should not use a global variable called `errno` for any purpose other than its standard use for returning error codes from the system. If `errno` is declared to be static, or `sizeof(errno) != 4`, it is not handled specially.

Compiler Options

The `vpcc` Command Line

Each file given on the command line must have a recognised suffix; the suffix identifies the file type:

Suffix	File Type
<code>.c</code>	C source code
<code>.cxx</code> , <code>.cc</code> , <code>.cpp</code> , <code>.C</code>	C++ source code
<code>.i</code>	Preprocessor output of C source code
<code>.ii</code>	Preprocessor output of C++ source code
<code>-</code>	Standard input (Preprocessing only).
<code>.h</code>	Header file (Preprocessing only).
<code>.s</code>	Assembler source code (unlinked)
<code>.asm</code>	Assembler source code (linked)
<code>.ism</code>	Library assembler source code (unlinked). May use <code>.s</code> instead.

Any other files are passed to the linker. These include:

<code>.o</code>	Object code (unlinked)
<code>.a</code>	Archive library, contains object code files.
<code>.so</code>	library description

Suffixes are recognised case sensitively, so `foo.Cxx` and `bar.CXX` are not recognised, `alpha.c` is a C source code file, and `baz.C` is assumed to be a file containing C++ source code.

The DFA utility is called after assembly if `-O` is specified. The default action is to insert zaps and remove redundant tags and unreachable code. Only fatal errors are output unless `-v -v` (twice) is specified. If debugging is specified then a check is made for wrong parameters in qcalls, and only zapping is done - redundant tags are not removed.

The environment variable `build.target` may be set to the absolute name of a directory within which a build is to be made. Any compilation with this set must be within the directory or a subdirectory. Its chief effect is to ensure that any toolname generated is relative to the build directory rather than the root. It also sets the default include and loader search paths to first look relative to the build directory and then to the absolute location. The names of input files are not affected, if a path is absolute it will be relative to the root rather than the build target so it is normally best to specify relative names.

Command line options

These are the command line options that `vpcc` understands. Many of these are standard GCC options and have the same meaning as their Linux or Classic Amiga ADE equivalents:

If a `.asm` file appears on the command line, most flags don't make sense since the file has already been linked and no other files can appear on the command line.

`-aa` Normally any comments in the `.s` file are stripped before the final `.asm` file is output. This option causes `vpas` and `vpld` to keep any comments so they can be seen in the `.o` and `.asm` files.

`-ansi` Disable any GNU features that might stop an ANSI conformant program compiling. The preprocessor defines `__STRICT_ANSI__` and the ANSI standard include files will only include ANSI functions.

`-asm` Suppress the assembling phase. Each source file is run through the preprocessor, translated, and linked with the others to produce `vpout.asm` or a `.asm` file named by the `-o` option.

`-build <directory>` The directory is assumed to be the build target relative to which all absolute file names or directories are found or created. If not specified then the contents of the shell variable `build.target` are used instead. If neither are present then absolute paths are not altered. Default directories are searched relative to the build target directory first and then via the absolute name.

`-B<path>` Use `<path>` as the base path for default include and library searches instead of the default.

`-c` Suppress the linking phase. Each source file is run through the preprocessor, compiled, and assembled to produce an unlinked object file. The resulting files have the same base name, but have suffix `.o`.

`-cp<tool path>` Override the tool directory inferred from any `-t` or `-o` parameter. It is better to specify the full tool name using `-t`.

`-cxx` Force C++ compilation. This flag is passed to the linker and is required when `vpcc` is given a set of `.s` files and therefore can't otherwise tell whether it's dealing with C or C++.

- dD
- dM
- dN Tell the preprocessor to output details about macro definitions.

- da
- dv Emit a VP2 specific RTL debugging dump with the file suffix .vp2.

- D <symbol> [=<value> Define preprocessor symbol. If no =<value> appears, then this has the effect of a #define <symbol> within each source file. If the =<value> appears, then the option has the effect of a #define <symbol> <value> within each source file.

- e <entry-point> Use the specified procedure as the entry point for the linked tool. This will override any explicit or default entry point e.g. main. This must be used if -T is specified and there is more than one procedure as optimisation may reorder the procedures.

- E Run only the preprocessor on the given C or C++ files. The output goes to standard output if -o is not specified. Standard input may be specified using -. If saving the output it is usual to name it the same as the input but with suffix .i (for C) or .ii (for C++), which can be compiled directly without preprocessing.

- f<name> Enable special option <name>. These options are used for debugging the compiler and other unusual tasks.

- g Generate directives for source code level debugging.

- gt An alternative way to enable the -mtrace-funcs flag. This flag is also passed to the linker.

- I<path> Add the given directory <path> to the search path used by the preprocessor to locate include files.

- l<libname> Include the shared library lib<libname>.so from the loader library list during the link stage.. If that does not exist the archive library lib<libname>.a is searched for instead. The default entries include -lc, and if a C++ file is compiled or -cxx is specified the -lcp also.

- L<directory> Add the directory to the loader library list. The loader library list ends with /lang/cc/lib, or <basedir>/lib if -B is specified.

`-m<name>` Enable special option `<name>`. These options are used for debugging the compiler and other unusual tasks.

`-M<opt>` Causes the preprocessor to output dependency information. Use only with `-E`. `-MM` omits dependencies for system includes.

`-n` Don't actually run any of the compiler components, but figure out what would need to be run. This is really only useful in combination with the `-v` flag.

`-nostdinc` Don't include the standard include file paths when preprocessing.

`-nostdinc++` Don't include C++ specific include file paths when preprocessing.

`-nostdlib` Don't include any standard libraries in the link.

`-o<name>` Specify the base name for the final result of compilation. This flag has no effect if a `.asm` file is on the command line. Using the `-o` flag in a context where more than one output file would be generated by the command is not allowed.

`-O`

`-O0`

`-O1`

`-O2`

`-O3` Select the level of optimisation. Optimisation is disabled by default and explicitly using `-O0`. The lowest level of optimisation is enabled by `-O` (equivalent to `-O1`).

The `-O2` flag optimises for size and performance. The compilation will likely take more time to complete, especially if there are very large functions. The highest optimisation level is enabled by `-O3`. When optimisation has been enabled, the preprocessor symbol `__OPTIMIZE__` is defined. Optimisation of C++ programs is not supported and may crash the compiler or generate incorrect code. Zapping is currently disabled during compilation; the DFA utility is called after assembly to insert zaps if optimisation is specified.

`-pedantic` Issue all the warnings required by ANSI standard C. This will warn about GNU extensions and some traditional C extensions that are not part of the standard.

`-save-temps` Do not delete any intermediate files. This option does not currently cause intermediate files to be placed in reasonably named files in the current directory like GCC. Add `-v` to see names of intermediate files generated.

`-sysdev` The linker will produce a tool that may be used as a device driver in elate. The global data is accessed via the environment pointer `__ep` and all gos calls are turned into `gose` calls passing `__ep`. The `-T` option must not be specified.

`-S` Finish after compiling to the unlinked assembler. The default is to name the output files the same as the input with the suffix replaced by `.s`.

`-SS<num>` Set the minimum stack size of the tool to be `<num>`. The default size should almost always be large enough.

`-T` Generate a single tool as the final output. This suppresses the output of a control object and alters the behaviour of the compiler and linker slightly. As a programmer's aid, this automatically defines the C preprocessor symbol `__TAOS_TOOL__` causing the `-ftaos-tool` flag to be passed to the compiler.

`-t<tool name>` Specify the full tool name to be used for the tool object. By default the tool name is derived from the output file specified by `-o`, the name is the terminal name less any `.asm` or `.00` and the directory is the output file directory relative to the build target directory. The terminal name only may be specified if `-cp` is used to specify the directory.

`-U<symbol>` Eliminate `<symbol>` preprocessor definitions in each source file.

`-v` Be verbose, displaying output indicating which programs are being executed and with what arguments. This flag can be repeated to increase the verbosity, so try `-v -v` if you want to know more about what's going on.

`-vp<num>` Generate version `<num>` of VP. At present, `<num>` must be 2, which is the default. This option is deprecated and may be removed.

`-w` Suppress all warning messages.

`-W<name>` All of the standard GCC warning flags are supported. For example, `-Wall` turns on all warning flags.

`-Wa, <option>` Pass `<option>` to `vpas` when producing unlinked object files.

`-Wl, <option>` Pass the `<option>` to the linker `vpld`.

Default options and paths

The option defaults for C are:

The `-fomit-frame-pointer` flag is set. Use `-fno-omit-frame-pointer` to override it.

The symbols `__ELATE__` and `__GNUC__` are predefined by the preprocessor.

Include path for the preprocessor:	<code>lang/cc/include</code>
Library path for the linker:	<code>lang/cc/lib</code>
Executable for the preprocessor:	<code>lang/cc/bin/vpcpp</code>
Executable for the translator:	<code>lang/cc/bin/vpcc1</code>
Executable for the unlinked assembler:	<code>app/stdio/vpas</code>
Executable for the linker:	<code>app/stdio/vpld</code>
Executable for the assembler:	<code>asm</code>
Executable for the DFA utility:	<code>dfa</code>
Output assembler name:	<code>vpout.asm</code>
Output tool name:	<code>vpout.00</code>
Run time support library:	<code>lang/cc/lib/libc.so</code>

The defaults for C++ are the same as for C except:

Executable for the translator:	<code>lang/cc/bin/vpcp1</code>
Run time support libraries:	<code>lang/cc/lib/libcp.so</code> <code>lang/cc/lib/libc.so</code>

The symbol `__cplusplus` is also defined for C++ source files.

Compilation examples

```
vpcc tmp/main.c
```

This command will cause the file `tmp/main.c` to be compiled, linked, and assembled. The resulting output will be in `vpout.00`.

```
vpcc -c tmp/main.c
```

The above command will cause the file `main.c` to be compiled and assembled. The resulting unlinked object file will be in `tmp/main.o`.

```
vpcc -o tmp/bar tmp/main.c
```

This command will cause the file `main.c` to be compiled, linked, and assembled. The resulting output can be found in `tmp/bar.00`.

```
vpcc -cp test tmp/main.c
```

That command will cause the file `main.c` to be compiled, linked, and assembled. The resulting output will be found in `test/vpout.00`.

vpld Options

The linker is responsible for combining files of C/C++ object code and library descriptions into one VP assembler file that is ready to be assembled using `asm`. Each of the `<file>` arguments on the command line is an object file from the C/C++ assembler `vpas`. Programmers typically use `vpcc` to compile and link, rather than running `vpld` directly.

The following command line flags are recognised by `vpld`:

Command line flags

`-cp <ctlpath>` Specifies `ctlprefix` and prefix for the toolname to be `<ctlpath>`.

`-cxx` Link using C++ conventions instead of those for C. This is generally unnecessary as output of the C++ compiler is identified by a pseudo-op in the file.

`-g` Emit source code debugging information.

- gt Generate tracing information in code generated by the linker.

- L <searchdir> Add the directory <searchdir> to the list of directories searched for a shared library or archive as specified using -l <library>. The earlier entries are searched first.

- l <library> Link using the shared library lib<library>.so ; or if that does not exist, use the archive lib<library>.a first found in the search path specified using -L <searchdir>.

- o <filename> Send output to <filename> instead of the standard output.

- O <level> Optimisation level 0..3 as for vpc, Default is zero , -O with no parameter sets level to 1. This option currently has no effect but may in the future cause extra work to be done to improve the code output.

- SS <size> Set the minimum stack size of the generated control object (default 262144 bytes) to <size>. Insufficient stack may cause a crash, so it is occasionally necessary to request a larger stack size.

- shared Output a shared library, the output should refer to a .so file to create which will contain the shared library details for use in further links.

- split Split by entry block into a directory. The main output will be a tool to initialise the data, and call the main entry point if it is not a shared library. The directory will be called <toolname>.dir.

- sysdev Output a tool that may be used as a device driver. The global data is accessed via the environment pointer __ep and all gos calls are turned into gose calls passing __ep.

- T Output a single tool.

- t <toolname> Set the name of the tool to <toolname>.

- v Verbose. Repeat to increase amount of detail.
- version Print the version of the linker and usage information and exit.

- vp<num> Identify the version of VP as <num>. This defaults to 2.

vpld pseudo-ops

Module Control

These pseudo-ops must in general appear in a module header except for `.linkonce`, which may also appear in a module body. A complete object file must have `.object` at the start and `.end` at the end.

```
.als amount
```

The amount given by all of these pseudo-ops is summed to help ensure that the program is given a reasonable stack allocation.

```
.constructor initializer
```

[C++ only] The initializer is added to the list of functions to be called at run time just before the main program is entered. They are called in reverse order to that in which the pseudo-ops appear.

```
.cplusplus
```

This indicates that the file was generated by the C++ translator and the program requires appropriate libraries, initialization, and termination.

```
.destructor terminator
```

[C++ only] The terminator function is added to a list of functions to be called when the program exits. The destructor functions are called in the same order as declared by the pseudo-ops.

```
.end
```

Every object file must end with this, as a check when reading archive members to avoid reading beyond the end. See `.module_start` for an example of use.

```
.entry name, params
```

This names the entry point after any initialisation has been done and specifies the parameters. This allows main to omit the parameters or return void. One and only one entry point may be defined, except if a pure elate tool is being produced - see `toolentry`. The entry point name should also have `.export` specified to make it public.

The standard parameters are p0 i0: i0 corresponding to

```
int main (int argc, char *argv[])
```

This can be defined as the entry point by:

```
.export !.main!,0
.entry !.main!,p0 i0:i0

!.main!:
.ent p0 i0:i0
```

The `.entry` line makes the conventional parameters explicit.

```
.header source-line
```

The source-line will be put by at the start of the assembler output file after any standard includes. The usual use would be to include additional assembler `.include` directives at the start. No quotes are necessary around source-line, it is simply the rest of the line.

```
.ident "string"
```

The string can be used to put identification information for the source file or compiler into the output tool. For example:

```
.ident "$GCC: (GNU) 2.8.1 $"
```

would identify the compiler used. The '\$' characters at either end are necessary if the string is to be recognised and output by the ident tool.

```
.library
```

This module is optional - it is only included if required by a non-weak external.

```
.link_off
.link_on
```

The lines between these are copied by `vp1d` exactly into the output assembler file except names within `!...!` have name mangling performed on them.

Linkonce blocks

```
.linkonce link-key [,check]
.linkonce_end
```

During linking only one module's linkonce blocks for each link-key will be linked.

The declaration and initialisation or code for a symbol must either not be in a `.linkonce` block or else must share the same link-key. No locals declared in a `.linkonce` may be referenced outside. An `.export` or `.extern` within a linkonce block may be referenced outside the block and will be treated as an `.import` or an `.extern` without initialisation if the linkonce block is not included.

The link-key is not a symbol name - it does not conflict with other names and may be the same as a function or data area name. The check is an alphanumeric string, which if specified must be the same for each particular key. It is enough to specify check once only in the header to get full checking as early as possible.

The functions and data declared in a linkonce block need not be marked with `.weak` to be removed, it is enough that they be part of a duplicate linkonce block.

If used a `.linkonce` will normally appear once in the module headers enclosing the linkonce declarations, and once in the module body enclosing the actual instances or initialisations.

Multiple modules

```
.module_start count
.module_body count
.module_end
```

There may be more than one module in an object file. The declarations are grouped at the start and the bodies put at the end as in:

```
.object
.module_start 1
.export !.abc!,i0:i0,1
.module_start 2
.extern !.def!,8,4,0,0
.module_body 1
!.abc!:
.ent i0:i0
```

```
        ret
.module_end
.end
```

The count starts at 1 in each file and connects a body to its declaration. The linker reads the declarations in its first pass and the module bodies in the second pass.

Module start

The following pseudo-ops must only occur in a `.module_start` block.

```
.als
.cplusplus.main.toolentry
Declarations pseudo-ops
Interface pseudo-ops
```

The following pseudo-ops may occur in either a `.module_start` or `.module_body` block.

```
.linkonce
.linkonce_end
.file
```

All other pseudo-ops may only occur in a `.module_body` block.

```
.object
```

The magic string `".object\n"` must be at the start of every `.o` or `.so` file or it will be rejected by the linker. See `.module_start` for an example.

```
.toolentry [name]
```

A **go** to the function name is put at the start of the output. This pseudo-op may only be used once per file when linking a pure tool - pure tools are ones without any initialisation or static data.

Declarations

These pseudo-ops may only occur in the `.module_start` part of an object file.

No locals declared in a `.linkonce` may be referenced outside. Any globals are treated as if they are referenced outside.

The **params** specification in the pseudo-ops below is used for cross-checking. It's an unique compressed form of an **ent** only mentioning the highest register and with the registers in the order i,l,f,d,p. Thus, **i2p0:-** means three integers and one pointer register on input and none on output.

The **xrefs** in the declarations counts the number of references to the symbol other than in diagnostic records or calls. This is used to help arrange the per-process data block for more efficient execution.

The **calls** in the declarations counts the number of calls using **gos** to the symbol.

```
.check name, check
```

The **check** is an alphanumeric string. Two instances of **.check** which specify the same name must also specify the same check.

```
.equate equate-name, basename, offset, xrefs
```

This declares that **equate-name** is a new name for the memory address that is the sum of symbol **basename** and **offset**. **offset** may be positive or negative. If **basename** is a function then so is **equate-name** and **offset** must be zero.

```
.extern name, size, alignment, initflag [,initstr], xrefs
```

This declares **name** to be a variable of length **size** bytes that requires its first byte to be on an address evenly divisible by **alignment**; if **alignment** is zero, there is no alignment requirement. The following table shows how **initflag** is interpreted:

- 0 No initialisation is supplied. A zeroed space in the per-process data-block will be allocated if no other **.extern** supplies an initialisation. The name is searched for in libraries if necessary but it is not an error if it is not found.
- 1 The variable's per-process data-block is initialised by a block copy instruction at start up time. The initial value is supplied in a **.data** block.
- 2 The **initstr** assembler instruction moves some value into **p0**. The value in the register is then moved into **name**. This sets up **FILE *stdin, *stdout, *stderr**;
- 3 The variable is a read-only const. **vpas** will have checked that any addresses are to other local read-only areas or labels.

4 The `initstr` is the name of an offset from `gp` in the `PROC` struct. This is used to set up an external to `errno`.

5 The `initstr` is used directly for name. This may be used to substitute an absolute number.

6 The `initstr` is an assembler instruction that moves some value into `p0`. This is the address of `name`; `initdlag 2` sets up a 4 byte pointer to the area.

7 No initialisation is supplied but the name is not searched for in libraries. A zeroed space is allocated if no other initialisation is found.

If `initflag` is ORed with 32 then it denotes a common area. There is no visible effect except that `initflag 32|0` is equivalent to `initflag 7`.

If there is more than one `.extern` for the same symbol then the size and alignment of the symbol is the maximum of those specified.

An `.extern` in a duplicate `.linkonce` is treated as if its `initflag` is zero.

If `initflag` is non-zero in more than one `.extern` a warning is output unless the symbol is marked `.weak` and only the first initialization encountered will be applied.

```
.export name, params, calls
```

This indicates that `name` is a global function that is defined in this module.

The parameters and calls are as described in the Declarations section.

An `.export` in a duplicate `.linkonce` is treated like an `.import`.

```
.fdesc name, xrefs
```

The function name is referenced to get the address of the function. This may require an entry in the data area to hold the value.

```
.function name, [params], calls
```

The name specified is that of a local function.

The parameters and calls are as described in the Declarations section.


```
.import name, [params], calls
```

This signifies that `name` is a global function that is not defined in this module - in other words, it is an external function that needs to be linked up to.

Once again, the parameters and calls are as described in the Declarations section.

```
.static name, size, alignment, initflag, xrefs
```

This declares `name` to be a variable of length `size` bytes, that requires its first byte to be on an address evenly divisible by `alignment`. If `alignment` is zero, there is no alignment requirement. `initflag` may be 0, 1 or 3 and the effect is as described for `.extern`. There is no implicit address ordering between successive `.static` variables.

```
.syspath directory-path
```

The `directory-path` is the path used for finding the tools specified in any following `.system` pseudo-ops. This applies till a subsequent `.syspath` or the end of this module.

```
.system name, [params|], calls [, pathnoep, pathep]
```

The `name` refers to a system tool in the directory given in the last `.syspath`. It is treated as a `.weak .export` defined in this module so a user defined version takes precedence. They require no environment set up when called. This is used to set up the initial C library.

The parameters and calls are as described earlier under the heading Declarations.

The `pathnoep` and `pathep` specify the full path name to be used for the tool depending on whether environment pointers are being used. They may be used to supply two different versions of `bsearch` or `qsort` depending on the form of procedure pointer being handed over, for example.

```
.weak name
```

The symbol `name` is marked with the `weak` property. This means that if `name` is an `.import` (unsatisfied external reference) then it is not an error if `name` is not fixed up. It will not be searched for in a library but will be fixed up if found. A non-weak unsatisfied reference overrides this behaviour.

If name is an `.export` (global function) then there may be more than one copy of name. A version which is not marked as weak takes precedence otherwise the first encountered is chosen.

If name is an `.extern` (global data) then more than one initialisation may be supplied and the first encountered will be chosen; a non-weak version takes precedence.

When linked in a shared library a weak `.export` or `.extern` will be referenced via an indirection so they can be linked instead to a version in the caller. In other cases references to external data that are satisfied in a shared library will be fixed up to by an indirection in the main program.

Data Initialisation Pseudo-ops

When using these, be aware that the data in `.rodata` blocks is considered preformatted whereas the initialisations in a `.data` block may be reordered.

```
.address name, offset ,dst-offset
```

This declares a data word containing the address of name+offset within the initialisation for a destination symbol in a `.data` block. The `dst-offset` determines the displacement after the destination symbol where the address goes.

```
.data
    . . . . .
!abc!:
    dc.i 1234
    .address !def!,12,4
```

This says !abc! is initialised to { 1234, &def+12 }

```
.align
```

This aligns the current address in a `.data` block to the next 8 byte boundary. The start of each symbol in a `.data` block is determined by the alignment in its corresponding `.static` or `.extern` and a `.align` before the symbol is ignored.

```
.data
```

This introduces a block of static initialisations: values that set the initial value of data declared by `.static` or `.extern`. This is ended by `.text` or `.rodata`.

```
.retool ref-type, "tool path", dst-offset
```

This is similar to `.address` except the pointer is initialised as in:

```
dc.p arh tool-path
```

`ref-type` is a tool reference type; 0 means a normal tool reference.

```
.rodata
```

This introduces a block of code-related data that is read-only and not described by `.static` or `.extern` statements. This will typically contain jump tables for computed branches. The only addresses allowed refer to local `.text` or `.data` blocks. These are set up using `dc.p !tag!` rather than `.address`.

Code Pseudo-ops

```
.debug statement
```

The statement following `.debug` is suppressed unless debugging is enabled. This enables `vpas` to conditionally remove unnecessary statements such as a **go** to an immediately following label.

```
.ent params
.entl params
.entd
.entdr
.entih
```

These are the entry points to functions and correspond directly to the VP assembler commands. The `.ent` and `.entl` will be converted into `ente` and `entle` commands if an environment pointer is required. Nothing special happens with the other forms but code after them will not be able to use `__ep`.

```
.gos target, params
```

This corresponds fairly directly to the VP assembler command **gos**. It may also become a **gose** or **qcall**.

If `target` is a pointer or expression, and environment pointers are in use, it will be the address of a procedure descriptor. The following code will be emitted:

```
gose [target], [target+4], params
```

When `target` is a symbol, the code emitted depends on the symbol. For a target within a main application a `gos` is emitted, and when linking inside a shared library the following code is emitted:

```
gose target, __ep, params
```

When linking to a `.system` symbol a `qcall` will be used.

When linking to a shared library or linking to an external from a shared library normally a `gose` will be used with a locally declared `.fdesc`.

Other forms like `ncall` or a version using `qcalle` may be supported in future.

```
.link off  
.link on
```

These two pseudo-ops are used to bracket text that is copied, without any change whatsoever, to the output stream. This capability might be used, for example, to include hand-crafted code that requires no linking and contains no other pseudo-ops. These directives may not be nested. The text is considered as in the code area.

```
.text
```

This starts a code block. The code blocks and `.link` blocks are concatenated in the output. It is possible that some or all initialisations from `.data` or `.data` blocks will also be inserted where they appear, so the code generation should not assume they are necessarily out of line.

Diagnostics Pseudo-ops

```
.file ``filename``
```

This identifies the source code file corresponding to this assembler file. Used by the linker when producing error messages and translated to the assembler's `__file` macro.

```
.line number
```

This identifies the source code line number corresponding to this point in the assembler file. It is translated to the assembler's `__line` macro.

```
.stabs "string", type, access, desc, value
.stabn type, 0, desc, symbol
.stabd type, 0, desc
```

These DBX/stabs debugging pseudo-ops are translated to the assembler's `__stabs`, `__stabn`, and `__stabd` macros. These will become a compatibility feature when DWARF format is supported instead.

Both `vpas` and `vpld` inspect and update the stabs. The string field may specify a global data item. If so, the value field will be set by `vpas` to the global symbol. The access field may be updated to say how the symbol is to be located by a debugger. The details of type definitions in the string field are not inspected.

If the string field end with `\\` it is assumed to be continued by the next stabs record. The `<name>:<type char>` part must fit into the first stabs record when it is split.

Some information from `.stabs` is kept from a duplicate `.linkonce` block, in particular debug information for globals and type information - this is done by adjusting stabs records to replace local symbol names by a single space and change the `.stabs` to be that for a type definition.

The access field in the output `__stabs` macros describes how a value may be used to get the location of a symbol, as follows:

- 0 standard action - `vpld` has not set the access specially
- 1 value is an absolute value or location.
- 2 value is a tag in the code area for code or constant data.
- 3 value is a displacement in the data area
- 4 value is a displacement into the data area where there is a
 pointer to the required location
- 5 value is a displacement into the data area where there is a
 pointer to a function descriptor for a code location.

Interface Pseudo-ops

These pseudo-ops may only occur in the `.module_start` part of an object file.

```
.interface
```

This module describes the desired appearance of the output after linking. Thus, any `.export` describes a name that must be visible in the output. Any `.weak` applies to the names after linking. No actual code or data blocks should be supplied. By default any `.global`, `.export`, and `.extern` anywhere in the link will be visible unless `.suppress` is used.

```
.require library
```

This is used in a shared library description to say that it uses another shared library called `library`. If the using library is included in a link then `library` is also automatically included. A name declared in `library` is not automatically available in the description of the using library unless it has a defining instance.

```
.retain name
```

This ensures that `name` is kept visible in an output library even if `.suppress` has been specified. The name must be present in the linked module. No other properties are specified.

```
.shared library
```

This says the current module is a description of a shared library and that `library` is the tool name for the library.

```
.suppress
```

No visible names will be retained in the output library unless they are explicitly specified in a `.library` module.

```
.version version, revision
```

This sets the version and revision level when a shared library is output. When a shared library is loaded at run time a check is made that it has the same version and a greater or equal revision to the one the module expects.

Compatibility Pseudo-ops

The following initialisation directives only appear if `vpas` uses compatibility pseudo-ops.

```
.setaddr src-name, src-offset, dst-name, dst-offset
```

This pseudo-op is used to initialise variables in the per-process data block that point to absolute addresses.

The address of `src-name` plus `src-offset` is copied into `dst-name` plus `dst-offset`.

```
.setref ref-type, "tool path", dst-name, dst-offset,
```

This pseudo-op is used to initialise a variable in the per-process data block that points to a function.

The address of the tool is copied into `dst-name` plus `dst-offset`. This variation of the pseudo-op is used to initialise a variable that is a pointer to a function.

More about Amiga C

The new Amiga is a truly portable operating system, in every sense. It is equally at home on the desktop, within an appliance, or tucked into a pocket. The underlying model is of a Virtual Machine or Processor for which all Amiga programs are written. Our unique translation technology takes the Virtual Processor byte code and translates it into native code of the target processor.

Normally, translation into efficient native code only takes place when loaded from backing storage such as a disk or network. The translator knows which processor it is running on and can generate the appropriate code. Programs for Amiga can currently be written either in the assembler language of this virtual processor, VP code, or in C, C++ or the Java language. Work is underway to define ingenious new Amiga-specific languages.

Portability

The Amiga system is shipped with a C compiler because the vast majority of applications that were intended to be portable have been programmed in that language, or the C++ variant. The Amiga compiler is based on GCC, the de facto standard compiler for cross-platform development. GCC also runs on Classic Amigas and a plethora of embedded, Unix and desktop systems.

If the bulk of your program is already written in C, GCC is an efficient way to port it. However you will probably still need to address the new features of the Amiga. In particular, programs that make assumptions about a 16 or 32 bit environment are likely to fall foul of the Amiga's full 64 bit capability. These are discussed in the 'troubleshooting' section of the *Amiga C Compiler* chapter.

The effort to resolve those issues is worthwhile, because it will make your programs more robust in the long run. If the source has already been written with architectural independence in mind, the changes should be slight. If not, they will be instructive, and a valuable learning experience for the future.

You will also need to consider the differences in user interface between the Amiga and the system you are porting from. The Amiga shell and desktop environment neatly provide all the usual interface facilities, though they differ in detail from Classic Amiga, Unix or other desktop systems.

We are working on a style guide for the new system. For the time being, you should use the interface of existing Amiga applications as a guide to the way you implement new ones. Consistency will improve the chance that your application will become an accepted, well-integrated part of the system.

C or C++ are commonplace, and important components of the Amiga system, but they are not necessarily the first choice for new developments. Java and VPcode potentially offer greater productivity and efficiency if you are developing new programs from scratch.

All of these languages are available from the start in our development system, because we do not believe that there is one ideal tool for any job. We have given you a choice of compilers, with more to come, to help you get the best results in any given application.

Re-usable Tools

There is also an opportunity to make programs more modular and reusable as you port them to the Amiga system. This will cost extra effort in the short term, but should pay dividends later.

Amiga is designed so that all application programs are written as small sections of code called **tools**, which are executable pieces of code. The whole of Amiga, including the kernel and all its functions, is programmed this way. This is called object-based programming and is supported by Amiga's unique use of Dynamic Binding technology.

An application is a tree structure of a number of dependent tools. Tools are **re-entrant** and **multi-threaded**, being discarded if they are no longer referenced and only if the memory is required. This ensures that Amiga is also extremely memory efficient.

Amiga already comes with over 6,500 tools coded and ready for use. These tools are typically less than 1K in size. They include kernel functions, ANSI C libraries and many others. All are available to the development programmer on the development platform. You will find it easy to develop more.

This chapter covers the main aspects of tool programming when writing in C. Other parts of this manual discuss Amiga Tool Programming in VP and Java.

Greeting the world

This page outlines how to compile and run a C program. The example prints the words 'Hello World' to the screen. The Amiga Operating System comes with example C programs in the directory `demo/example/c/`.

Type this at the Amiga command prompt to compile the program `hello.c` :

```
$ vpcc demo/example/c/hello.c -o demo/example/c/world
```

The instruction to invoke the compiler is `vpcc`. The compiler is then given the name of the source program that it is to compile, including its full pathname. The shell checks for programs in `/app/stdio` by default. Generally Amiga commands are typed while in the root directory, with paths given relative to the root. Compiling must take place from the root directory.

The option `-o` specifies the output tool name. If a destination path is not specified, the default location is `'/'`.

The line above invokes the compiler to compile the program `demo/example/c/hello.c`, telling it to output the tool to the same directory location and to name the tool `world`. Therefore, if there were no errors, the compiler will have created a separate file named `world.00`. This is the executable file. To run that program, type this at the command prompt:

```
$ demo/example/c/world
```

The words "Hello World" are then printed to the screen.

In the next section the elements that make up the program `hello.c` are explained in detail.

Building and using tools with C

In the C source code it is possible to call any Amiga tool, regardless of the language that was used to code that tool originally, such as C, C++ or VP. Whether written in C or C++, compiled tools can be called from VP source code. In all cases source code is compiled to a tool in VP binary format.

The tool calling mechanism is a powerful facility for making Amiga tool calls appear as regular functions. Because tools are not regular functions, some constraints exist on such things as memory allocation, as we shall soon see.

Most Amiga tools already have the calling mechanism enabled through existing header files. The Amiga reference material clearly documents these C interfaces, if they already exist. It is only necessary to use the special declaration syntax explicitly for new tools that are to be called from within the C coded tool or application.

Declaring Tool Calls

Tool calls are declared using a modified version of the `__attribute__` mechanism of GNU C. The `__attribute__` mechanism differs from regular GNU C in that the attribute parameters are strings instead of identifiers. To declare a C function as a tool call you must specify the following:

- The full C declaration of the function, including the return type and parameters
- The assembler instruction to call the tool

For example, the declaration of the `lib/acos` tool looks like this:

```
extern double acos(double x) __attribute__((qcall lib/acos));
```

This indicates that a function call to `acos()` should be mapped to the tool invocation `qcall lib/acos`, with the argument being placed in register `d0` and the result returned in that register.

The compiler determines the assignment of function arguments to registers, but in rare circumstances it is necessary to explicitly state which registers are used to pass the arguments. To override the default tool parameter passing convention, a tool call may look something like this:

```
extern int func(arg1, arg2, . . . , argn) __attribute__((
    "qcall */atool",
```

```

        "args reg1,reg2,. . .,regn" /* MUST BE PROVIDED. */
    ));
extern int func(arg1, arg2,. . .,argn) __attribute__((

```

This indicates that a function call should be mapped to the tool invocation `qcall */atool`, with the first argument (from left to right) being placed in register `reg1`, the second in `reg2` and so on. The return value is the default set by the tool.

Although rarely necessary, it is also possible to override the default return register:

```

extern int func(arg1, arg2, ... argn) __attribute__ ((
    "qcall */atool",
    "args reg1,reg2,. . .,regn" /* MUST BE PROVIDED. */
    "return reg1"                /* MUST BE PROVIDED. */
));

```

Functions with variable arguments are supported, as in this declaration of the `printf()` tool:

```

extern int printf(const char *format, ...)
__attribute__(( "qcall lib/printf" ));

```

Using Tools in C++

It is possible for C++ code to call tools but the tool call declarations must be explicitly declared as C:

```

#ifdef C++
extern "C" { . . .
    qcall . . .
}

```

C++ to Assembler Translator

The C++ translator, which is called `vpcp1`, is similar to the C translator but processes C++ source code instead of C. It is a heavily customised version of the standard GNU C++ translator.

Once declared, a tool is called much like a regular C function. Note that the name of the tool is similar to the name of a C function in that it is in effect a function pointer. The following example illustrates this idea :

```
#include <stdio.h>
extern double sqrt(double x)
    __attribute__((qcall lib/sqrt));
extern double fabs(double x)
    __attribute__((qcall lib/fabs));

double (*arr[])(double) = {sqrt, fabs};

int main(void)
{
    double x, y;

    x = arr[0](144.0);          /* sqrt() */
    y = arr[1](-x);            /* fabs() */

    printf("Result should be 12: %f\n", y);
}
```

Creating qcallable tools in C

Amiga already has many useful tools or functions that can be called from C programs, but application-specific tools may still have to be created. To create a qcallable tool in C, the tool C source code must be compiled to VP byte codes, there should be a header file available to specify the qcall attribute and there must be a calling program.

To illustrate this, a tool is coded in C that defines a function that, after compilation, is a qcallable tool. For the `sayhello.c` program to print to the screen, the Amiga tool `lib/printf` must be called. A parameter `hello_string` has also been defined which will be printed by `lib/printf` when it is used in another program. The source of this tool can be found in `demo/example/c/sayhello.c` :

```
extern int printf(const char* format, ...)
    __attribute__((qcall lib/printf));

void sayhello(char *hello_string)
{
    printf ("%s\n", hello_string);
}
```

Before this tool can be used by our next program it must be compiled and a VP tool created. To compile `sayhello.c` enter this at the Amiga shell command prompt:

```
$ vpccl demo/example/c/sayhello.c -o demo/example/c/sayhello -T
```

As before, the `-o` option tells the compiler the name that the tool is to be given. When calling this tool in another program, the name of the tool and its directory given at this point are the ones that must be specified at the `qcall`. `-T` tells the compiler that it is a secondary tool, as discussed later in this chapter under the heading **Compiler Options**.

For a tool to be used in a program, the compiler must know where to find it. A header file is therefore created. A header file tells the compiler of any parameters or return results and the name of the tool to be called. The header file ensures that any functions or data defined is invoked in the way specified as opposed to the conventional C function calling convention.

In our example, the header is called `sayhello.h`, and the tool already coded and compiled is called with a `qcall` to `demo/example/c/sayhello`. Note that the function name `sayhello` is also defined:

```
extern void sayhello(char *) __attribute__((
    "qcall demo/example/c/sayhello"
));
```

When the header file `sayhello.h` is included in any program and a line of code which says `sayhello` is inserted, the compiler will know to emit `qcall demo/example/c/hello, (p0:-)`. This example program is called `testsayhello.c`:

```
#include <stdio.h>
#include "sayhello.h"

int main(int argc, char *argv[])
{
    sayhello("this is hello_string !");
}
```

Here is a simple program which calls a user-defined tool to perform a mathematical operation. The first program defines the function, the second is the header file and the third is the actual program calling the newly defined function.

demo/example/c/mathtest.c

```
int mathtest(int x,int y)
{
    return((x*y)-1);
}
```

demo/example/c/mathtest.h

```
extern int mathtest (int x,int y) __attribute__
((
    "qcall demo/example/c/mathtest"
));
```

demo/example/c/usemath.c

```
#include <stdio.h>
#include "mathtest.h"

int main(int argc,char *argv[])
{
    int mathtest_result;
    printf("qcalling mathtest...\n");
    mathtest_result=mathtest(5,6);
    printf("mathtest(5,6) returns %d\n",mathtest_result);
}
```

Compile both of the .c files like this ; the -o part is a continuation of the line:

```
$vpcc demo/example/c/mathtest.c -T
      -o demo/example/c/mathtest
$vpcc demo/example/c/usemath.c -o demo/example/c/usemath
      -o demo/example/c/usemath
```

This will appear when the demo/example/c/usemath program is run:

```
--
qcalling mathtest...
mathtest(5,6) returns 29
--
```

The tool's entry point is determined by the first function linked in. In the case of a single C file, it will be the first function listed in that file.

Object-based style

The Amiga system allows applications to be designed and written in an object-based style. Amiga object-based programming is built around small tools which are dynamically loaded into RAM just when they're needed.

Design issues

Two fundamental design issues must be addressed before object-based coding can begin. The programmer needs to consider:

- The definition of the instance data structure for the object, where it is to be defined, how it is to be allocated and initialised and how it is to be de-initialised and de-allocated when no longer required.

In traditional Object-Oriented Programming terms this is comparable to declaring class member variables, also known as class properties. The initialisation and de-initialisation code is comparable to the code typically found in constructor and destructor methods of traditional Object classes.

- Whether all the method code should be written within the class or whether it would be beneficial to dynamically load a tool when the method is actually called, thereby minimising the amount of memory required when first loading the class into memory. The tool is called using either the `VIRTUAL` or `VIRTUAL+FIXUP` options to achieve this.

It is normally the responsibility of the application using the object to allocate and initialise the instance in line with the policy defined by the class programmer. If the object has already been allocated and initialised then the application is not responsible for this. For instance a device driver object may have been allocated and initialised when the system was first booted.

Memory Structure

The definition of the instance data structure for the class instance can be defined either within the class itself or as an include file. If the structure is defined within the class, it will be difficult to inherit from that class. It is therefore recommended that the structure be defined in an include file which is then referenced at the beginning of the class source file. A baseclass structure's first offset must always be defined as OB_SIZE.

A subclass structure must begin at the size of the baseclass from which it is inheriting.

It is also necessary to check whether an include file has already been included elsewhere. This is to avoid redefinition errors if multiple source files are included in the same structures. A similar technique was employed in the Classic Amiga OS include files.

Class include files

In the following examples the class properties include parameters to hold the location and size of a buffer area.

baseclass code

```
.if ~?def(BASECLASS_SIZE); ensure BASECLASS_SIZE defined
structure OB_SIZE ; must always use OB_SIZE in base class
int32 PROPERTY1; properties of class
int32 PROPERTY2
int32 PROPERTY3
size BASECLASS_SIZE; size of this base class structure
#endif
```

subclass code

```
.if ~?def(SUBCLASS_SIZE) ; include baseclass properties
#include 'demo/example/baseclass/class'
structure BASECLASS_SIZE;offset baseclass structure size
int32 SUBCLASS_PROPERTY1
size SUBCLASS_SIZE ; size of subclass structure
#endif
```

Instance allocation and deallocation

The allocation and deallocation of an instance of a class is the responsibility of the application. Objects are usually allocated memory via calls to an appropriate library function like `sys/kn/mem/allocdata`. Typically, the programmer writes an allocator tool called `_new`, whose main task is to allocate memory for the object. There will also be a corresponding de-allocator tool, called `_delete`, which frees up memory allocated to the object. These special tools are normally coded in the same source file as the class methods.

In summary, the programmer will code the class properties in a `class.inc` file. The class methods will be coded in a file called `class.asm`, which will also include the special allocator and de-allocator tools, `_new` and `_delete` respectively. Each class must have its own sub-directory because of this naming convention.

Defining a Class and Method Coding

Within the Amiga system a class is a distinct type of tool with particular macros provided to build the contents. The name of the class is defined immediately after the first macro class. The name is the full pathname of the class tool. By convention, the name of the class tool is **class** within the same directory as the `_new` and `_delete` tools.

The **classend** macro defines the end of the class. All the methods for the object's services including the initialisation, de-initialisation methods and defaultmethod are coded within these two macros.

method is a macro which defines a named service of the class. The macro is immediately followed by a subroutine containing code for the service. This may be all the code required or it may be a call to an external tool. If the method code is held as a tool, it can be coded in the same source file as the class, although it will be held separately on disk once assembled.

A subroutine does not have to end with a **ret**, though it has to end with something which does not permit execution to fall through it. This could be any one of **ret**, **go**, **endloop** (for a loop which does not contain a break), **parentclass**, **chain**, or **chainclass**. Also note that **noret** is allowed; this is an indication that the call just before it is guaranteed not to return, for example when it throws an error and does not return). For more information on these terms, please see the *VP Reference Manual* which is supplied as a PDF file on your Development CD.

Example baseclass method framework:

```
; demo/example/baseclass/class.asm
.include 'taort'
.include 'demo/example/baseclass/class'

class 'demo/example/baseclass/class',VP

method _init
    ent p0:i0
    tracef "baseclass : _init\n"
    clr [p0+BASECLASS_PROPERTY1]
    clr [p0+BASECLASS_PROPERTY2]
    clr [p0+BASECLASS_PROPERTY3]
    clr i0    ; return 0 if OK
    ret

method _deinit
    ent p0:i0
    tracef "baseclass : _deinit\n"
    cpy 1,i0 ; drivers use <0 error, 0= in use, 1 as OK
    ret

defaultmethod
    entd
    tracef "baseclass : default method called.\n"
    ret
classend
```

Two methods, `_init` and `_deinit`, must be defined in the class code as well as a `defaultmethod`. The `_init` method initialises the object instance and the `_deinit` method performs the opposite of `_init`. All other methods are private to the particular class.

The `defaultmethod` defines the code to be executed if a method called is not provided. The method name called by the application must match exactly the method name defined within the class, otherwise the `defaultmethod` will be called instead. The `defaultmethod` can code for an error.

```
defaultmethod
    entd
    ; method name not recognised handler code
    ret
```

If the default method of a subclass is called, control is transferred to the baseclass default method. The baseclass is also known as the parent class.

```
defaultmethod
entd
parentclass      ; call baseclass default method
```

defaultmethod is not always necessary when its parent class is the only requirement. If it is included, a ret will not be required, as above. When defaultmethod is not included, the system will provide one. If this happens to be a subclass, the system version will provide a parentclass. If it is a baseclass, the system will execute a noret which will produce undefined behaviour, except when running with the pentiumt configuration, when it will produce a breakpoint trap.

Another method type that may be defined within the class file is **xmethod**:

```
:
xmethod char
ent p0 p1 i0 i1 i2 i3 i4 i5 i6:i0 i1
qcall lib/grf/fnt/chardraw,
      (p0,p1,i0,i1,i2,i3,i4,i5,i6:i0,i1),VIRTUAL|FIXUP
ret
:
```

This lets you make a call like this:

```
ncall p0,@char,(p0:p3)
```

A call of this type (with @) returns a pointer to the method code, in this case in p3. The method code can thus be invoked subsequently, without the overhead of a normal ncall, by using a **gos** instruction, thus:

```
gos p3,(p0,p1,i0,i1,i2,i3,i4,i5,i6:i5,i9)
```

It is also still possible to use a normal ncall to invoke the method code.

Allocation and De-allocation of Memory

For the methods of an object to be accessed by an application, an instance of the class of the object must be in memory. An allocation and de-allocation policy must be decided upon by the application programmer but conventionally the class programmer will provide a tool to allocate memory, called `_new`, and a tool to

deallocate, called `_delete`.

There are several Amiga system functions available for memory allocation, but for this example we will use `sys/kn/mem/allocdata`. It is preferable to use `sys/kn/mem/allocdef` for memory allocation in device drivers as this allocates memory from a pool that can be shared by multiple processes and persists once the allocating process has closed. The ANSI library functions are not usually used in this situation as they may not be available to device drivers.

Allocating memory using the `_new` tool

Memory allocation is carried out by a tool because an instance of the object has not yet been created. However, the source of the `_new` tool can be coded within the same source file as the class but is held separately once assembled, and which is stored in the same directory as that of the object.

The `_new` tool allocates the memory, and it must also reference the class and return the instance pointer in `p0`. To allocate a memory block of the correct size, we need to allocate the size of the instance data structure as defined in the instance include file.

```
cpy BASECLASS_SIZE,i0
qcall sys/kn/mem/allocdata,(i0 : p0,i0)
```

Use the macro **refclass** to initialise the object header and reference the class with the full name of the class tool. `refclass` searches for the class tool, bringing it into memory and translating it only if it is not already present on the tool list.

```
refclass p0,demo/example/baseclass/class
```

Allocator code

```
; allocator tool in demo/example/baseclass/class.asm

tool 'demo/example/baseclass/_new',VP,0
ent - : p0      ; return instance pointer
cpy.i BASECLASS_SIZE,i0
qcall sys/kn/mem/allocdata,(i0:p0,i0)
if.p p0 != NULL
    refclass p0,demo/example/baseclass/class
endif
ret
toolend
```

Deallocating memory using the `_delete` tool

The `_delete` tool reverses the effect of the `_new` tool provided by the class programmer. The `_delete` tool can be coded within the same source file as the class but it is held separately once assembled. It is stored in the same directory as that of the object.

The `_delete` tool de-references the class, using the macro **derefc**lass, and then frees the memory.

Deallocator code

```
; de-allocator tool in class.asm
tool 'demo/example/baseclass/_delete',VP,template
ent p0 : -
derefc p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend
```

Initialising and deinitialising instance data

Once an object has been allocated memory and referenced, the application has access to the methods within the class. The instance data structure should be initialised by calling the `_init` method, before any of the other methods will be available for use. Otherwise, member variables in the instance data structure will not be in an initialised state.

The coding of the `_init` method of a class will depend on the type of object that is to be initialised. It may take the form of private memory allocation or setting up values within all or part of the instance data structure.

If initialisation should fail, a tidy up routine must be carried out by the application to delete the object.

If the instance being initialised is a subclass, the baseclass section of the instance data structure must be initialised first. This is effected by calling the baseclass `_init` method. If this is successful, the subclass section can then be initialised.

If the subclass `_init` method was unsuccessful, but the baseclass `_init` was successful, the baseclass `_deinit` method must be called before returning an indication of failure to the application.

Only after initialisation has been successful is the instance of the class - the object itself - available for use by the application.

Once an object is no longer required the instance must be deinitialised before it can be deleted. The `_deinit` method must be a complete reversal of the `_init` method and is again private to the object. Once deinitialisation has been successful the instance may be deleted.

Defining a Baseclass

The definition of a class, be it a base or sub class, requires that the correct include files are defined at the beginning of the class source file. These are normally the standard Amiga `taort` include file, and any private include files specifying the structure of the instance data.

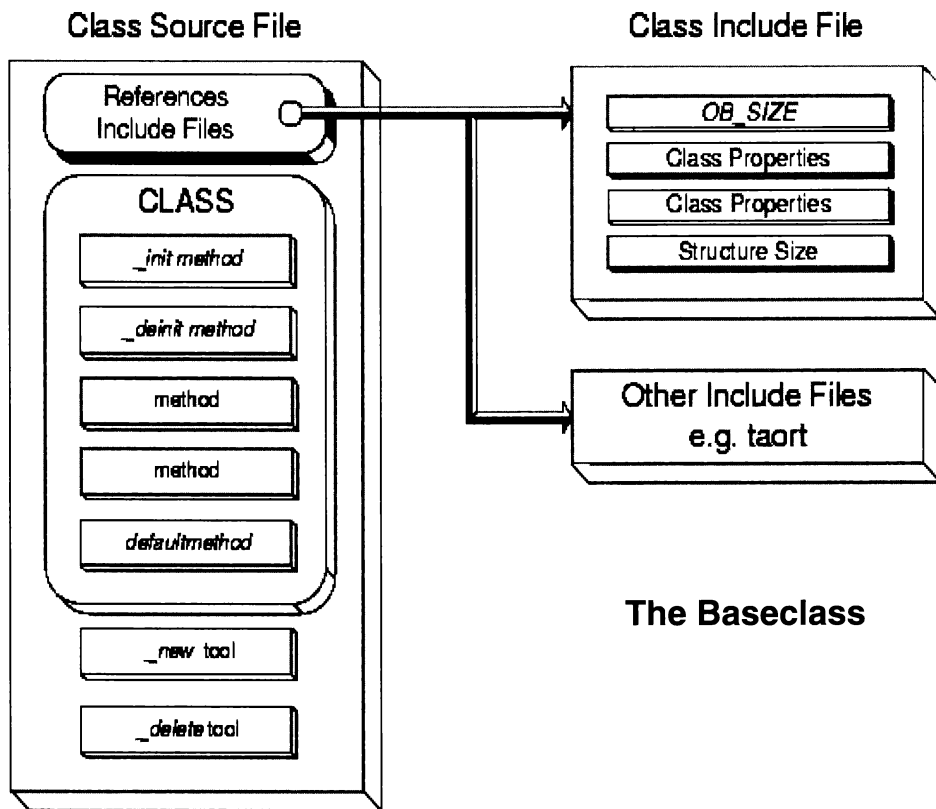
If the instance data structure is not to be defined within an include file it can be defined within the class source file, before the class macro. However, it should be remembered that if the instance data structure is defined inside the class source file, it will not be possible to inherit from the class easily.

Once the include files and instance data structure have been defined, these must be immediately followed by the class macro as described earlier.

The two compulsory methods, `_init` and `_deinit`, are defined next. Like all methods, the `_init` and `_deinit` methods can make external calls to tools, for greater memory efficiency.

```
method _init
    ent p0:i0
    tracef "baseclass : _init\n"
    clr [p0+BASECLASS_PROPERTY1]
    clr [p0+BASECLASS_PROPERTY2]
    clr [p0+BASECLASS_PROPERTY3]
    clr i0 ; return 0 if OK
    ret
```

The defaultmethod is coded after all the other methods. The defaultmethod must come after all other methods in order for the class to be created correctly. The code within the defaultmethod is specific to the class, as are all other methods, but it will normally be coded to log an error or throw an exception when in a baseclass. Sometimes the baseclass method will be written simply to return and perform no other action.



To close the definition of the class tool, use the **classend** macro.

Baseclass code

```
; class.asm --baseclass
.include 'taort'
.include 'demo/example/baseclass/class'

class 'demo/example/baseclass/class',VP

method _init
ent p0:i0
tracef "baseclass : _init\n"
clr [p0+BASECLASS_PROPERTY1]
clr [p0+BASECLASS_PROPERTY2]
clr [p0+BASECLASS_PROPERTY3]
clr i0 ; return 0 if OK
ret
```



```

method _deinit
    ent p0:i0
    tracef "baseclass : _deinit\n"
    cpy 1,i0 ; drivers use <0 error, 0= in use, 1 as OK
    ret

method getval
    ent p0:i0
    tracef "baseclass : getVal\n"
    cpy [p0+BASECLASS_PROPERTY1],i0
    ret

method setval
    ent p0 i0 : -
    tracef "baseclass : setVal\n"
    cpy i0,[p0+BASECLASS_PROPERTY1]
    ret

defaultmethod
    entd
        tracef "baseclass : default method called.\n"
    ret
classend

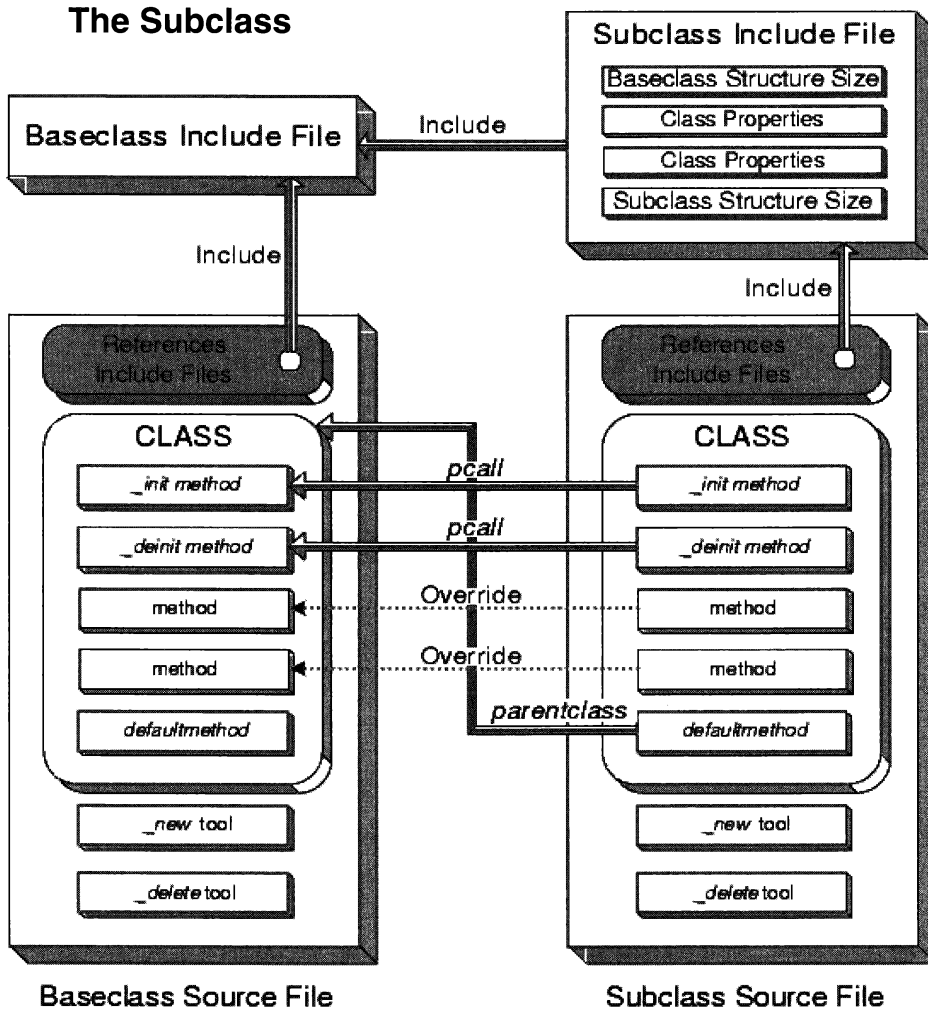
tool 'demo/example/baseclass/_new',VP,template
    ent - : p0 ; return instance pointer
    tracef "baseclass : _new\n"
    cpy.i BASECLASS_SIZE,i0
    qcall sys/kn/mem/allocdata,(i0:p0,i0)
    if.p p0 != NULL
        tracef "referencing base class\n"
        refclass p0,demo/example/baseclass/class
    endif
    ret
toolend

tool 'demo/example/baseclass/_delete',VP,template
    ent p0 : -
    tracef "baseclass : _delete\n"
    derefclass p0
    qcall sys/kn/mem/free,(p0 : -)
    ret
toolend
.end

```

Defining a Subclass

A subclass is defined in a similar manner to a baseclass with some alterations, as shown in this figure:



The subclass name is defined after the class macro, with the name of the baseclass added before the language definition. Notice that there are no quotes around the baseclass name.

```
class 'demo/example/subclass/class',  
      demo/example/baseclass/class,VP
```

Subclass methods

The `_init` method is still required, but the subclass `_init` method must now include a call to the baseclass `_init` method to initialise the baseclass section of the instance data structure so that its methods are also available for use. This is achieved by using `pcall` (parent call) within the subclass `_init` method. Note that a `pcall` can only be made from within a class tool, unless a large method's code is moved into a separate tool, in which case the programmer may use the `__extends` macro at the start of that tool to enable the use of the `pcall` macro in that tool.

```
pcall p0,_init,(p0:i0)
```

The `_deinit` method of the subclass must also call the baseclass `_deinit` method by using `pcall` after de-initialising the subclass section of the instance data structure.

```
pcall p0,_deinit,(p0:i0)
```

The `defaultmethod` must pass responsibility to the baseclass as a method called and not found may be within the baseclass. The command for this is `parentclass`.

```
defaultmethod
entd
; additional code here if required
parentclass
ret
```

All other methods are private to the subclass. Any methods defined in the subclass, with the same name as methods defined in the baseclass, will override such baseclass methods. To close the definition of the subclass code, the `classend` macro is used, as before.

Subclass code

```
; class.asm -- subclass

.include 'taort'
.include 'demo/example/baseclass/subclass/class'

class 'demo/example/baseclass/subclass/class', \
    demo/example/baseclass/class,VP

method _init
ent p0:i0
```

```

    tracef "subclass : _init\n"
    pcall p0,_init,(p0:i0)      ; init baseclass
    ifnoterrno i0
    ; baseclass init was OK, init subclass instance data
    cpy 456,[p0+SUBCLASS_PROPERTY1]
    endif
    ret

method _deinit
    ent p0:i0
    tracef "subclass : _deinit\n"
    pcall p0,_deinit,(p0:i0)   ; base class deinit
    ret

method scmethod
    ent p0 i0:-
    tracef "subclass : subclassmethod.\n"
    cpy i0,[p0+SUBCLASS_PROPERTY1]
    tracef "[p0+SUBCLASS_PROPERTY1] = \
    %d\n",[p0+SUBCLASS_PROPERTY1]
    ret

defaultmethod
    entd
    parentclass
    ret
classend

tool 'demo/example/baseclass/subclass/_new',VP,0
    ent - : p0
    ; return instance pointer
    cpy.i SUBCLASS_SIZE,i0
    qcall sys/kn/mem/allocdata,(i0:p0,i0)
    if.p p0 != NULL
    refclass p0,demo/example/baseclass/subclass/class
    endif
    ret
toolend

tool 'demo/example/baseclass/subclass/_delete',VP,0
    ent p0 : -
    derefclass p0
    qcall sys/kn/mem/free,(p0:-)
    ret
toolend
.end

```

Calling an object from within an application

For an application to be able to call a method of an object, it is necessary for an instance of the class to be in memory. This can be achieved by the application calling the `_new` tool of the class required.

```
qcall demo/example/baseclass/_new, (-:p0)
```

This QCALL returns the object reference.

In some circumstances, memory for the object will be allocated from within the application, in which case the application must reference the class by using the **refclass** macro and create the instance, in the same manner as the `_new` tool.

If the object was successfully created `p0` will be the instance pointer, and the object can then be initialised.

The named call

To call a method we use the Amiga named call, `ncall`.

```
ncall p0,_init,(p0:i0)
```

If the initialisation was successful, all of the methods of the object will be available to be called. The application has no knowledge of whether the object instance is a subclass or a baseclass and has no interest in how the services are provided by the object. All the methods can be called by using `ncall` and the specific method name. If the method name is not called correctly or is not available, the defaultmethod will be invoked.

Once the object is no longer required, it is the job of the application to deinitialise and delete the instance in line with the policy defined, calling the `_deinit` method, and then either calling the `_delete` tool or de-referencing the class and freeing the instance memory.

Baseclass test

```
; test.asm
.include 'taort'

tool 'demo/example/baseclass/test',VP,F_MAIN,8192,0

ent - : -

    cpy.p 0,p1

; new instance
qcall demo/example/baseclass/_new,(-:p0) ; p0 object ref
if p0==0
    tracef "out of memory creating 1st baseclass object\n"
    go tidyup
endif
ncall p0,_init,(p0:i0)
iferrno i0
    tracef "error initialising 1st baseclass object\n"
    qcall demo/example/baseclass/_delete,(p0:-)
    cpy.p 0,p0
    go tidyup
endif

; another new instance
qcall demo/example/baseclass/_new,(-:p1);p1 object ref
if p1==0
    tracef "out of memory creating 2nd baseclass object\n"
    go tidyup
endif
ncall p1,_init,(p1:i0)
iferrno i0
    tracef "error initialising 2nd baseclass object\n"
    qcall demo/example/baseclass/_delete,(p1:-)
    cpy.p 0,p1
    go tidyup
endif

cpy.i 4,i0
ncall p0,setval,(p0 i0:-)

cpy.i 123,i0
ncall p1,setval,(p1 i0:-)
```

```

clr i1
ncall p0,getval,(p0 : i1)
tracef "getval returned : %d \n",i1

clr i1
ncall p1,getval,(p1 : i1)
tracef "getval returned : %d \n",i1

ncall p0,fred,(-:i0)

tidyup:
if p0!=0
; Deinitialise and destroy first object
ncall p0,_deinit,(p0:i0)
qcall demo/example/baseclass/_delete,(p0:-)
endif
if p1!=0
; Deinitialise and destroy first object
ncall p1,_deinit,(p1:i0)
qcall demo/example/baseclass/_delete,(p1:-)
endif

; shutdown
qcall lib/exit,(0:-)
ret
toolend
.end

```

Subclass test

```

; test.asm
.include 'taort'

tool 'demo/example/baseclass/subclass/test', \
      VP,F_MAIN,8192,0

ent - : -

; new instance
qcall demo/example/baseclass/subclass/_new,(-:p0)
if p0==0 ; p0 object ref
tracef "out of memory creating subclass object\n"
go tidyup
endif
ncall p0,_init,(p0:i0)

```

```

    iferrno i0
        tracef "error initialising subclass object\n"
    qcall demo/example/baseclass/subclass/_delete, (p0:-)
        cpy.p 0,p0
        go tidyup
    endif

    cpy 4,i0
    ncall p0,setval, (p0,i0:-)

    clr i1
    ncall p0,getval, (p0:i1)
    tracef "returned by method : %d \n",i1

; this method not defined
    ncall p0,fred, (-:-)

; defined subclass method
    ncall p0,scmethod, (p0,i0:-)

tidyup:
    if p0!=0
        ncall p0,_deinit, (p0:i0)
        qcall demo/example/baseclass/subclass/_delete, (p0:-)
    endif

; shutdown
    qcall lib/exit, (0:-)
    ret
toolend
.end

```


File names and paths

Generally in the Amiga system the class hierarchy is reflected in the directory hierarchy. Subclasses should be located in directories below the parent class. For the example used in this manual the directory structures should be:

```
demo/example/baseclass/class.asm
```

```
demo/example/baseclass/subclass/class.asm
```

If this structure was used a call to create a subclass object would be:

```
qcall demo/example/baseclass/subclass/_new, (-:p0)
```

In other words, the directory structure should be reflected throughout the class definitions. This approach was not used in this manual to help keep the names more manageable.

For the latest versions of the code referred to in this manual please consult the directory `demo/example/`.

Format of listings

In this and other examples in this volume, program lines that are too long to fit this page in a readable font have been split - a bold backslash symbol "****" appears at the end of the line that continues below.

In a few cases long lines have been dragged to the left of their normal indented position, as this is less untidy than splitting the content of a line that is otherwise just too long to fit, and is considerably cheaper than supplying a document with fold-out page extensions, like some of the Classic Amiga system schematics!

A Classy example

The following example demonstrates the principles that have been discussed in this chapter. Four classes are used. In the example: node, job, list and queue. Their functionality is reminiscent of basic routines in the Classic Amiga Exec library, or Knuth's *Fundamental Algorithms*. There is also a simple test program.

Node class include file

```
; demo/example/node/class.inc

.if ~?def(NODE_SZ)
    structure OB_SIZE
        int32 ITEM
        pointer NEXT
        pointer PREV
    size NODE_SZ
.endif
```

Node class code

```
; demo/example/node/class.asm
.include 'taort'
.include 'demo/example/node/class'

class 'demo/example/node/class',VP

method _init
ent p0 i0: -
cpy i0,[p0+ITEM]          ; initialise node
cpy.p NULL,[p0+NEXT]
cpy.p NULL,[p0+PREV]
ret

method _deinit
ent p0 : -
cpy 0,[p0+ITEM]           ; deinitialise node
cpy.p NULL,[p0+NEXT]
cpy.p NULL,[p0+PREV]
ret

method get_node_item
ent p0 : i0
cpy [p0+ITEM],i0
ret
```

```

method get_node_next
ent p0 : p0
cpy.p [p0+NEXT],p0
ret

method set_node_next
ent p0 p1: -
cpy.p p1,[p0+NEXT]
ret

method set_node_item
ent p0 i0 : -
cpy i0,[p0+ITEM]
ret

defaultmethod
entd
    tracef "node class default method.\n"
ret
classend

tool 'demo/example/node/_new',VP,template
ent - : p0
;tracef "debug: creating new node ...\n"
cpy NODE_SZ,i0
;tracef "debug: NODE_SZ = %d\n",i0
qcall sys/kn/mem/allocdata,(i0 : p0 i~)
refclass p0,demo/example/node/class
ret
toolend

tool 'demo/example/node/_delete',VP,template
ent p0 : -
derefclass p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend
.end

```

Job class include file

```

; demo/example/job/class.inc
; inherits from node
.if ~?def(JOB_SZ)

```

```

.include 'demo/example/node/class'
structure NODE_SZ
    int32 PRIORITY
size JOB_SZ
.endif

```

Job class code

```

; demo/example/job/class.asm -- sub-class of node

.include 'taort'
.include 'demo/example/job/class'
.include 'demo/example/node/class'
class'demo/example/job/class',demo/example/node/class,VP

method _init
ent p0 i0: i0
pcall p0,_init,(p0 i0 : -)
cpy 0,[p0+PRIORITY]
ret

method _deinit
ent p0 : -
pcall p0,_deinit,(p0 : -)
cpy 0,[p0+PRIORITY]
ret

method get_job_pri
ent p0 : i0
cpy [p0+PRIORITY],i0
ret

method set_job_pri
ent p0 i0 : -
cpy i0,[p0+PRIORITY]
ret

defaultmethod
entd
    parentclass
ret
classend

tool 'demo/example/job/_new',VP,template
ent - : p0
cpy JOB_SZ,i0

```

```

qcall sys/kn/mem/allocdata,(i0 : p0 i0)
refclass p0,demo/example/job/class
ret
toolend

tool 'demo/example/job/_delete',VP,template
ent p0 : -
derefclass p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend
.end

```

List class include file

```

; demo/example/list/class.inc

.if ~?def (LIST_SZ)
; list header
structure OB_SIZE
    pointer HEAD
    pointer TAIL
    int32 COUNT
size LIST_SZ
.endif

```

List class code

```

; demo/example/list/class.asm

.include 'taort'
.include 'demo/example/list/class'

class 'demo/example/list/class',VP

; methods

method _init
ent p0 : -
; init list structure
cpy.p NULL,[p0+HEAD]
cpy.p NULL,[p0+TAIL]
cpy.i 0,[p0+COUNT]
ret

```

```

method _deinit
ent p0 : -
ncall p0,zap_list,(p0 : -)
ret

method add_node
; adds node object to end of list
; p0 is list object
; p1 is node object to add
ent p0 p1 : -
if.p [p0+HEAD] = NULL
    cpy.p p1,[p0+HEAD] ; update head
    cpy.p p1,[p0+TAIL] ; tail and head point to same node
endif
cpy.p [p0+TAIL],p2 ; p2 is tail
ncall p2,set_node_next,(p2 p1 : -)
cpy.p p1,[p0+TAIL]
cpy.p NULL,p2
ncall p1,set_node_next,(p1 p2 : -)
inc [p0+COUNT]
ret

method print_list
; p0 is list object to print
ent p0 : -
clr i0
clr i1
cpy.p [p0+HEAD],p1
tracef "list contains\n"
while.p p1 != NULL
    ncall p1,get_node_item,(p1:i1)
    tracef "item %d:%d ",i0,i1
    ncall p1,get_node_next,(p1:p1)
    inc i0
endwhile
tracef "\ndebug: count is %d items\n",[p0+COUNT]
tracef "debug: count is %d items\n",i0
ret

method zap_list
; p0 is list object to zap
ent p0 : -
clr i0
clr i1
cpy.p [p0+HEAD],p1
tracef "zapping list...\n"

```

```

while.p p1 != NULL
    ncall p1,get_node_next,(p1:p2) ; save p1->next
    ncall p1,get_node_item,(p1:i1)
    tracef "debug: deleting node containing %d\n",i1
    ncall p1,_deinit,(p1:-)
    ncall p1,_delete,(p1:-)
    cpy.p p2,p1
    inc i0
endwhile
tracef "debug: %d nodes deleted.\n",i0
ret

defaultmethod
entd
    tracef "List class default method.\n"
ret
classend

tool 'demo/example/list/_new',VP,template
ent - : p0
cpy LIST_SZ,i0
qcall sys/kn/mem/allocdata,(i0 : p0)
refclass p0,demo/example/list/class
ret
toolend

tool 'demo/example/list/_delete',VP,template
ent p0 : -
derefclass p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend
.end

```

Queue class include file

```

; demo/example/queue/class.inc
; inherits from list

.if ~?def (QUEUE_SZ)
.include 'demo/example/list/class'
structure LIST_SZ ; same as list at present
size QUEUE_SZ
.endif

```

Queue class code

```
; demo/example/queue/class.asm
; Code TPB 98

.include 'taort'
.include 'demo/example/queue/class'

class'demo/example/queue/class',demo/example/list/class,VP

; methods

method _init
ent p0 : -
pcall p0,_init,(p0 : -)
ret

method _deinit
ent p0 : -
pcall p0,zap_list,(p0 : -)
ret

method add_job
ent p0 p1 : -
; p0 queue
; p1 job to add
clr i0
clr i1
cpy.p [p0+TAIL],p4      ; save tail
ncall p1,get_job_pri,(p1 : i1)
if.p [p0+HEAD]=NULL      ; list empty add to head
    ncall p0,add_node,(p0 p1:-)
else
    if.p p4 != NULL
        ncall p4,get_job_pri,(p4:i0)
        if i0 >= i1
            ncall p0,add_node,(p0 p1:-) ; add on end
        else
            cpy.p [p0+HEAD],p2
            ncall p2,get_job_pri,(p2:i0)
            if i0 < i1
                ; add before current head
                ncall p1,set_node_next,(p1 p2:-)
                cpy.p p1,[p0+HEAD]
                inc [p0+COUNT]
            else
```



```

        while.p p2 != NULL
            ncall p2,get_job_pri,(p2:i0)
            if i0 < i1
                ; insert job
                ncall p1,set_node_next,(p1 p2:-)
                ncall p3,set_node_next,(p3 p1:-)
                inc [p0+COUNT]
            endif
            cpy.p p2,p3 ; save
            ncall p2,get_node_next,(p2:p2)
        endwhile
    endif
endif
endif
endif
ret

method despatch_job
ent p0 : -
; not coded
ret

defaultmethod
entd
    parentclass
ret
classend

tool 'demo/example/queue/_new',VP,template
ent - : p0
cpy QUEUE_SZ,i0
qcall sys/kn/mem/allocdata,(i0 : p0)
refclass p0,demo/example/queue/class
ret
toolend

tool 'demo/example/queue/_delete',VP,template
ent p0 : -
derefclass p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend
.end

```

Like other examples in this book, this program appears on the Amiga Development CD in the demo/example directory.

Developing Tools

This chapter explains how you go about making and using Tools - the fundamental software building blocks of the Amiga system.

Assembling a Program or Tool

To run an application or to call a tool, it must have first been assembled. This section of the manual covers how to assemble and run a 'hello world' program. The Amiga Operating System comes with a number of example programs, and these can be found in the directory `demo/example/`, as can the source file of 'Hello World'.

All Amiga source files must be suffixed with `.asm` to denote that they are source programs. When they are assembled to `.00` byte code files they are unbound and in template form until they are loaded.

Hello to ASM

To assemble the program `hello.asm`, type the following command at the Amiga shell prompt while in the root directory:

```
$ asm demo/example/hello
```

If there were no errors, the assembler will have created a separate file with the name `hello`, suffixed with `.00`, containing VP byte codes. To run this program, type at the command prompt, in the root directory :

```
$demo/example/hello
```

The words "Hello World" are printed to the screen.

Note that the full pathname is required. The suffix is optional for `asm` files, whereas no `.00` suffix should be specified when running a tool, as you can see from the preceeding example.

Example Tool Source

The source code of 'hello.asm' follows. Every element required for programming a tool is in this program and is explained in the rest of this chapter.

```

.include 'taort'
tool 'demo/example/hello',VP,TF_MAIN,1024,0
ent - : -
printf "hello world\n"
qcall lib/exit,(0:-)
ret
toolend
.end ; the end of source directive is optional

```

It is instructive to look at the disassembly of this code. This can be achieved by:

```
$ dis demo/example/hello.00 | less
```

Disassembly is piped to the `less` utility to allow scrolling of the output.

Structure of Application Source Files

An application source file consists of an optional list of include files, the primary or standalone executable tool and any non-primary tools. Non-primary tools coded in the same source file are held separately when assembled and they need not be located in the same directory as the primary tool. On assembly, the tools will be suffixed with `.00` for tools coded in `VP`.

Include Files

Include files define the equates (constants) and macros needed for an application program. An include file's default extension is `.inc`. All system include files can be found in the default include file location: `lang/asm/include/`.

`taort.inc` defines standard system items for `VP` programs and is the most commonly used include file that is not application-specific.

The `.include` directive is used to specify an include file with the name of the include file placed directly after it within inverted commas. The assembler automatically prefixes this location if no path name is specified and suffixes `.inc` unless another extension or `"."` is specified.

```
.include 'taort'
```

Application-specific include files are absolute if prefixed with a `"/`. File names with no path specified, such as `taort`, come from the system include directory

/lang/asm/include/. Names with a path, like myinclude/foobar, are absolute, taken relative to the root directory whether or not they have a leading "/".

```
.include '/demo/example/foobar'
```

Application-specific include files may also be relative, in which case the assembler searches from the current directory location. As a rule, the assembler should normally be run from the root directory. This is not strictly necessary, but is suggested as good practice because in the long run it avoids confusion.

```
.include 'myinclude/foobar'
```

The Primary Tool

All application programs comprise one or more tools. The primary or main tool is the tool that is able to start a thread. It is this thread which executes non-primary tools.

Tools are referenced by name; this name is defined after any include files or defined global variables, but before the `ent` directive and code for that tool. As well as defining the tool name, it is also necessary to define the Assembler Language, the Tool Type, the Stack Size and the Global Variable Size. These are all specified on the same line, separated only by commas.

The tool name macro

The name by which the tool is to be referenced is positioned after the tool macro, which is placed within single quotes, '`<pathname>/<name>`'. Be warned that *tool names are case-sensitive*. If two tools are given the same name, differing only by case, they are two distinct and different files.

The Assembler Language

Tools must define the assembler language in which they are written. This can either be VPCode, native for PPC, or whatever, and must be defined for each tool.

The Tool Flags

Each application or program has one tool defined as the main or executable tool. This is indicated by using `TF_MAIN`. Once the program has been assembled, the name of the main tool is then the executable file.

The Stack Size

The stack size in bytes which is to be allocated is application-dependent but a minimum of 8,192 bytes is recommended. Should the stack size be insufficient it can be changed by the programmer. Also, although 8K bytes is the suggested initial stack size, Amiga has some support for dynamically extending the stack at run time if it turns out not to be big enough.

The Global Variable Size

The size of the global variables space (in bytes) which must be allocated for each instance of the application. Zero is permitted. The global variable space can be the size of a structure defined before the tool macro.

Example uses of the tool macro for primary tools

```
tool 'demo/example/hello',VP,F_MAIN,8192,0
tool 'demo/example/hello',VP,F_MAIN,16384,16
tool 'demo/example/hello',VP,F_MAIN,65536,GLOBALS_SIZE
```

The toolend macro

When all the code for the tool has been written, after the final `ret` in the code, and any data which may follow it, the end of the region started by the tool macro must be closed by the `toolend` macro. As it is possible to have more than one tool in a source file, the programmer has the option to make the end of the file more explicit by the use of an `.end` directive to conclude the file. Note that this is entirely optional as the assembler will finish processing on reaching the end of the source file.

```
ret
toolend
.end ; end of source directive - optional
```

Accessing Command Line Parameters

Sometimes it is necessary for a tool or program to access the command line input parameters. This is achieved by the use of the tool `lib/argcargv`. This tool returns C-style `argc` and `argv` parameters. It is standard practice for all primary tools to have a call to `lib/argcargv`, although that's not always required.

The `lib/argcargv` interface requires no inputs but outputs are a pointer to the block of argument pointers to the parameters (`argv`), which are fragments of the original command line string, and an integer to hold the number of arguments (`argc`). Parameter 0 is always the name of the program. The call to `lib/argcargv` is made directly after the `ent` directive and before any application specific code:

```
qcall lib/argcargv, (-:p0 i0)
```

Tidying up before closing a tool

Once all the code for the tool has been written it's wise to carry out some general housekeeping procedures. The Amiga tool `lib/exit` automatically performs these tasks. This tool works like the C language `exit()` function.

`lib/exit` automatically releases allocated memory blocks, and automatically closes files opened with `lib/fopen`. The standard I/O buffers are flushed and the tool's `stdin`, `stdout` and `stderr` channels are also closed.

`lib/exit` takes an integer to return to the shell, which is typically zero to indicate success. There are no outputs from `lib/exit`. Although it is not obligatory to make a `qcall` to `lib/exit`, it is strongly recommended that a call be made before closing the tool with `ret` and the `toolend` macro. In order to return successful status to the shell, `i0` is cleared before the `qcall` to `lib/exit`.

```
clr i0
qcall lib/exit, (i0:-)
```

This can be achieved even more simply like this:

```
Qcall lib/exit, (0:-)
```

Non-primary tools

Non-primary tools are tools that are only called from other tools. They can be coded in the same source file as the primary tool. The parameters `TF_MAIN`, stack size and global variables are not defined. A tool file will be created, suffixed with `.00` because it is coded in VP code. In all other respects coding of non-primary tools is the same as a main or primary tool. The equated constant "tool 'foo', VP" denotes a non-primary tool.

```
tool 'demo/example/hello2', VP, template'
```

Inside the Virtual Processor

Register Files

VP code liberates programmers from limited numbers of registers. Each tool or subroutine has its own set of five register files. Each file contains a specific type of data. These may be pointers, 32 or 64 bit integers, IEEE floats or doubles.

The `ent` directive at the beginning of a tool or subroutine specifies the total number of register inputs and outputs required by the piece of code being written. This is done for every tool or subroutine, effectively making an infinite number of registers available to the developer programmer.

Integer Registers

Integer registers may contain integers, bytes or characters. Their names start with `i0`. The default size for integer registers is 32 bits. All VP implementations support 32 bit integers, and the assembler assumes 32-bit integer format for all instructions unless they are otherwise qualified, and sometimes even then.

The VP architecture defines one special purpose register, `SI`, the signature integer. This special integer register is only used when making an `NCALL` to a method of an object. Application programmers should leave it alone.

Pointer Registers

Pointer registers are nominally 32 bits wide, like integer registers. All VP implementations support 32 bit pointers.

In addition to the general purpose pointer registers there are four special pointer file registers. Special registers are used in any place where ordinary registers are used, but care must be taken as other instructions may alter them implicitly. These special registers are `SP`, `GP`, `LP` and `PP`, and they are discussed in more detail at the end of this chapter.

The `ent` directive

The number of registers and the type required are specified at the beginning of a non-primary tool or routine. All tools and routines must have an entry directive as their first instruction before any other code. The standard entry point is `ent`.

Parameters are passed by any registers in each register file, with parameters appearing in low number registers in the respective files. Register files are local to a routine and unless they are explicitly passed to the called routine, they will not be visible in that routine and will be preserved when control is returned to the caller.

Parameters passed in and out must be specified on the entry instruction. Primary tools with the TF_MAIN bit set have an ent directive, which does not have any registers for input nor for output as the routine is self contained. All other tools must specify exactly which registers are to be used. All registers used must be contiguous starting from 0. The parameters are separated by a ':' with the inputs first and the outputs second.

A single '-' denotes no parameters, so a primary tool ent directive would be:

```
ent -:-
```

A non-primary tool ent directive might be:

```
ent p0 p1 i0 i1:i0
```

Local registers need not be specified, as this is taken care of automatically by the assembler. No other instructions should be placed before the ent directive.

Special Entry directives

Three more entry directives are available for special circumstances. You don't normally need to bother with these, but it may be useful to know that they exist.

entl is a variant only for leaf tools and subroutines - those that call no other tools or subroutines.

entih is a special kind of ent for interrupt handlers.

entd is reserved for the defaultmethod in object programming.

VP Instructions

Instruction basics

Instruction parameters can take three specific forms: constant, register or expression. Instructions are encoded as a single byte followed by parameters. One of the most commonly used instructions is `cpy`, which is used for register and memory access as well as moving constants into registers or memory.

The qualifiers for instructions of this kind are shown in the following table. A qualifier is only required in cases where registers other than 32 bit integers are employed, and the assembler is unable to automatically determine which form of register is required. It is also important to ensure that data in memory is appropriately aligned; in particular, this also applies to structures. The correct alignment for each data type, with examples of how qualifiers are used, is documented in the right hand column.

<code>.b = byte</code>	<code>cpy.b [p0],i0</code>	alignment: 1 byte
<code>.s = 16 bit short</code>	<code>cpy.s [p0],i0</code>	alignment: 2 bytes
<code>.i = 32 bit integer</code>	<code>cpy.i [p0],i0</code>	alignment: 4 bytes
<code>.l = 64 bit long integer</code>	<code>cpy.l [p0],l0</code>	alignment: 8 bytes
<code>.f = 32 bit float</code>	<code>cpy.f [p0],f0</code>	alignment: 4 bytes
<code>.d = 64 bit double float</code>	<code>cpy.d [p0],d0</code>	alignment: 8 bytes
<code>.p = 32 bit pointer</code>	<code>cpy.p [p0],p2</code>	alignment: 4 bytes
no qualifier	<code>cpy [p0],i0</code>	alignment: 4 bytes

To copy the contents of `p0` into `p1`:

```
cpy.p p0,p1
```

To copy the contents in the memory address pointed to by `p0` into `i0`:

```
cpy [p0],i0
```

To copy the contents of `i0` over contents of the memory location pointed to by `p1`:

```
cpy i0,[p1]
```

To copy the constant 42 into `i0`:

```
cpy 42,i0
```

To copy 99 into the memory location pointed to by the pointer p0:

```
cpy 99, [p0]
```

Get the idea?!

Return of the Guru

There is also a non-aligned version of the cpy command. To see how this works examine the following code, assuming p1 and p0 are integer-aligned:

```
cpy [p0], [p1+1]
```

This statement would cause the Amiga system to execute a hard-coded breakpoint when running the checking translator version of the build under the control of **ebug**, and possibly a system crash if you are using the non-checking translator version of the system. It's the equivalent of an address error, Guru Meditation code \$80000003, on a Classic Amiga. This is because the destination is not integer aligned; it's likely that the programmer actually wanted to do the following:

```
cpy [p0], [p1+4]
```

- that code would run perfectly. If a non-aligned copy were actually required then the correct code would be:

```
cpy.ni [p0], [p1+1]
```

Because there is a requirement that longs are long-aligned, and there is also a `cpy.nl` instruction to manipulate unaligned longs. Both the `cpy.ni` and `cpy.nl` operators are likely to decrease efficiency, using more bus bandwidth.

Expressions

More complex operations are generated by expressions. Expressions may contain expressions within themselves. Most VP operations are performed by expressions that are copied using the cpy instruction to a destination register. By using an expression on the left, much more complicated operations can be performed:

To add 6 to i1 and to place the answer in i3:

```
cpy (i1 add 6), i3
```

Expressions themselves can have other expressions within them. Therefore, to multiply i2 by 6 and then add to i1, placing the answer in i3, would be coded as follows:

```
cpy (i1 add(i2 mul 6)),i3
```

The assembler recognises the usual arithmetic symbols within expressions:

```
+ add
- subtract
* multiply
/ divide
```

The precedence of the operators in expressions matches that in the C language.

Macros are provided for some mathematical operations such as add:

```
add 6, i0
```

The above line generates the following code:

```
cpy (i0 add 6),i0
```

qcall, go, gos and ncall

qcall, go, gos and ncall are four ways of transferring execution to another section of code.

qcall

The qcall macro takes the name of a tool as a parameter, followed by the input and output registers. The programmer specifies appropriate registers for each individual application or routine. Local registers need not have been specified at the beginning of the tool.

```
qcall lib/argcargv,(-:p0 i0)
```

The tool is loaded and bound when the application referencing it is loaded. It will remain available in local memory for at least as long as the referencing application is in memory. It will not be relocated in memory whilst the caller is present.

Qcalls come in three distinctive flavours.

```
qcall demo/mytool, (p0:i0) ; non-virtual
qcall demo/mytool, (p0:i0), VIRTUAL ; virtual
qcall demo/mytool, (p0:i0), VIRTUAL+FIXUP ; semi-virtual
```

They may be non-virtual

This means that the tool was loaded and bound when the object referencing it was loaded. It will remain available in local memory at least as long as the referencing object is in memory. It will not be relocated in memory whilst the caller is present.

They may be virtual

This means that the required tool need not be in local memory at the time it is required. If the tool is not available in local memory the tool is loaded and bound before it is available for use. On exit from the tool its memory space may be deallocated by the kernel in order to free local memory space. While the tool is referenced it will remain in memory, but not necessarily otherwise.

They may be semi-virtual (VIRTUAL+FIXUP)

This means that, in the same way as a virtual tool, it is only loaded on the first call to it, and not when the caller is loaded - but is then treated as non-virtual, remaining in memory until no longer referenced by the calling tool.

ncall

`ncall` is used to call a named method of a class. The input and output registers must be specified.

```
ncall p0,drink, (p0:i0)
```

go and gos

`go` transfers execution unconditionally to a label within the same `ent` block, so no register passing is required.

```
go next_routine
```

`gos` transfers execution to a label placed just before the start of a different `ent` block. In this case, it is necessary to specify the registers to be passed.

```
gos sub_procedure, (inputs:outputs)
```

Labels or Tags

Labels in VP are tags which identify a location. Non-data tags are referenced by `go` and `gos` instructions:

```
gos go_subprocedure
```

Data tags are referenced by `cpy`.

```
cpy [data_label], i0
```

Tags are specified with a trailing colon `:`. Data tags are labels in the data section, which is introduced with the `.data` directive `data`.

```
str_label:    dc.b "some bytes",0
.align       ; ensures word alignment
data_label:   dc.i 12,34,56
```

Note the use of the `.align` directive to ensure that data is correctly word-aligned.

Structures and Memory Allocation

The VP assembler supports structures in order to simplify defining and accessing variables. Structures can be used to define the format of global and local variables, as well as allocated memory blocks.

A structure definition does not reserve memory space anywhere but only defines the names, offsets and size of a structure. When some memory of the given size is allocated, the fields can be referred to by their names, rather than by offset.

There is a balance between having 'packed' structures, which save space, and aligned integers, which may provide better performance). When the structure has been packed, without any padding to ensure integers are integer aligned, `cpy.ni` and `cpy.nl` should be used, with a consequent decrease in performance on some target platforms. If structure is not defined many times in your application, it's usually best not to pack it, especially if it is often used.

It is vital to use `nint32`, or another corresponding directive, to define an `int32` at an unaligned offset. Using `int32` or another corresponding directive in cases where the next offset is unaligned will generate an error. All the structure macros `int16`, `int32`, `int64`, `float32` and `float64` also have `n` prefixed versions.

Longs and doubles should precede integers and floats in mixed structures. These should be grouped together, by decreasing size, so that anything that is aligned to eight byte boundaries comes before anything that is four aligned, and so on. This is because `malloc` memory blocks are initially 64 bit long-aligned, and then proceed linearly onwards.

Global Variables

A global structure is defined before the tool macro to allow the variables to be managed by the kernel. We define the structure to begin at offset 0 (blank). The variables required are listed, specifying the type and the name. `size` names the current structure offset.

```
structure
    int32 NAMEANY1_A
    int32 NAMEANY1_B
    int32 NAMEANY1_C
size NAMEANY1_SIZE
```

In the tool definition the size of the structure would be `NAMEANY1_SIZE`.

```
tool 'demo/example/hello',VP,F_MAIN,8192,NAMEANY1_SIZE
```

The named global variables can now be accessed by using `gp`, the globals pointer.

```
cpy 1,[gp+NAMEANY1_A]
cpy 2,[gp+NAMEANY1_B]
clr [gp+NAMEANY1_C]
```

Local Variables

Local structures are defined inside the tool. We define the structure to begin at offset 0 (blank) and list the variables required specifying the type and the name. `size` names the current structure offset.

```
structure
    int32 NAMEANY2_A
    int32 NAMEANY2_B
    int32 NAMEANY2_C
size NAMEANY2_SIZE
```

The structure can then be allocated memory from the stack inside the tool :

```
allocstruct NAMEANY2_SIZE
```

This allocates NAMEANY2_SIZE bytes on the stack.

The structure is accessed by using the stack pointer, SP. However, if more than one structure is to be allocated on the stack, you must preserve the stack pointer and use standard pointer registers instead. To allocate a structure using a pointer register:

```
allocstruct NAMEANY2_SIZE,p0
allocstruct NAMEANYOTHER_SIZE,p1
```

To free space allocated on the stack by allocstruct use freestruct n, where n is the number of bytes to free. Note that allocstruct and freestruct pairs must be matched and nested. The exception to this is at the end of subroutines, where it is permissible to leave out the freestruct instruction - this works because ret frees up any memory allocated on the stack. It is also possible to use library functions such as lib/malloc and lib/free to allocate memory.

Allocated memory blocks

Memory blocks can be allocated using library functions such as lib/malloc or the underlying kernel functions like sys/kn/mem/allocdata.

By way of example, consider a simple buffer, declared as a global structure :

```
structure
    pointer BUF_START
    int32 BUF_SZ
size BUF
```

As the above structure is global, there is no need to specifically allocate memory for it. It is necessary to allocate memory for the buffer itself and its size may well be decided at run time. To specify the size of the buffer, for example 1K:

```
cpy 1024,[gp+BUF_SZ]
```

To allocate memory for the buffer:

```
cpy [gp+BUF_SZ],i0
qcall sys/kn/mem/allocdata,(i0 : p0 i0)
```

The returned pointer is then stored in the global variable for later use:

```
cpy.p p0,[gp+BUF_START]
```

When the buffer is no longer required, the memory can be deallocated thus :

```
cpy [gp+BUF_START],p0
qcall sys/kn/mem/free,(p0 : -)
```

Alternatively, the memory could have been allocated using `lib/malloc`:

```
qcall lib/malloc,(i0 : p0)
```

Freeing the memory block after calling `lib/malloc` is done by making a `qcall` to `lib/free`. However, you can take advantage of the call to `lib/exit` at the end of the tool and let this free the memory blocks the C library allocated instead.

Bit structures

It is possible to create bit maps or bit field structures in VP. These make use of the `dbitstart` and `dbit` assembler macros. `dbitstart` indicates the start of the structure and `dbit` defines the name of the actual bit for ease of reference in the code. For example:

```
.
dbitstart      ; mask,bit
    dbit FLAG0,BFLAG0
    dbit FLAG1,BFLAG1
    dbit FLAG2,BFLAG2
    dbit FLAG3,BFLAG3
    dbit FLAG4,BFLAG4
    dbit FLAG5,BFLAG5
    dbit FLAG6,BFLAG6
    dbit FLAG7,BFLAG7
```

After that example is assembled the mask names would have these values:

Mask name	Value	Bit name	Value
FLAG0	1	BFLAG0	0
FLAG1	2	BFLAG1	1
FLAG2	4	BFLAG2	2
FLAG3	8	BFLAG3	3
FLAG4	16	BFLAG4	4
FLAG5	32	BFLAG5	5
FLAG6	64	BFLAG6	6
FLAG7	128	BFLAG7	7

This macro helps a lot when performing bitwise operations on data. It is the equivalent of the `BITDEF` macro familiar from Classic Amiga include files.

Program Control

Loops

There are several types of loop construct in VP. They are implemented as macros. In the following notes the [.x] indicates the data type that is being tested such as .p for pointer. If the type is integer .i is optional as it's the default.

while .. endwhile

```
while[.x] <condition is true>
:
endwhile
```

Note that a test occurs before entering the loop so the code inside the loop may or may not run.

for .. next

```
for [<register>|<constant>,<register>]
:
next <register>
```

The for macro sets up a label at the start of the loop. If applicable it will also copy a constant or the contents of another register into the counter register. The <register> can be integer or long. The next macro will decrement the counter register contents and jump to the label set up by the for macro, as long as the contents of the counter register is greater than zero.

repeat..until

```
repeat
:
until[.x] <condition>
```

In this case the body of the repeat loop is always entered at least once.

loop-endloop

```
loop
  if i1=1
    break;
```

```

endif
:
breakif i0 = 0
endloop

```

This forms an unconditional loop. To exit the loop you must use a break or breakif statement. It is also possible to control loop iteration through use of the continue statement. This will cause the loop to jump to the next iteration of the loop, but does not necessarily terminate the loop.

Conditional code

VP includes high-level constructs to support conditional code execution. As before [.x] is the data type being tested and this is optional for integers.

if-elseif-else-endif

```

if[.x] <condition>
:
elseif[.x] <condition>
:
else
:
endif

```

switch-whencase-otherwise-case-endswitch

SWITCH is an elegant way of coding alternation - choice from a list of possibilities. The basic syntax of a SWITCH follows this outline:

```

switch
whencase <condition1>
whencase <condition2>
otherwise
:
break
case <condition1>
:
break
case <condition2>
:
break
endswitch

```

An example, `switch.asm`, is given at the end of this chapter. It's worth bearing in mind that a computed jump, indirected through a table, is often more efficient than the repeated tests within a long switch sequence. However it's also much more likely to go wrong if the value being tested falls outside expected limits...

bool macro

The `bool` macro provides a convenient way of performing a conditional jump. The basic syntax is:

```
bool <condition>,label
```

For example :

```
bool i0=0,exit
```

Rolling your own Macros

VP lets programmers develop custom macros to boost their productivity. The general format of a macro is:

```
.macro <macroname>  
; body of macro  
.endm
```

This is a trivial example of a macro:

```
.macro mymacro  
    .check %n = 3  
    clr %3  
    add 3,%1  
    add 3,%2  
    add %1,%3  
    add %2,%3  
.endm
```

The macro could then be called from VP code, like this:

```
mymacro i0,i1,i2
```

In this case 3 would be added to `i0` and `i1`. These two registers are then added together with the result placed in `i2`.

Observe the use of the `.check` directive. This checks the calls to the macro at assembly time and validates the number of parameters passed. Should a call have an incorrect number of parameters an assembler error message will be displayed.

Register Defines

It is possible to designate labels to registers using the `defbegin-defend` construct. This can often make code more readable. This example illustrates the syntax:

```
defbegin 0
    defp my_ptr
    defi an_integer
    defi a_byte
; code using the above register names
defend
```

In the above examples the label `my_ptr` is allocated to `p0`, `an_integer` is allocated to `i0` and `a_byte` is allocated to `i1`. Code can then manipulate the defined labels as if they were registers e.g. `cpy 2, an_integer`. On the `defend` the registers will correctly be deallocated and can then be used if required in the normal way.

It is also possible to have nested `def` blocks. Note the use of `0` after the `defbegin`. This indicates that this is the outer `def` block, or top level. This can provide an extra safeguard in complex programs as if the programmer tries to define another outer block, with `defbegin 0`, the assembler will give an error.

Nested blocks arise where you want to use some registers temporarily and would like the assembler to deallocate the registers so that they can be used elsewhere in the program. The following example illustrates this:

```
defbegin 0
    defi nVar1
    defp pVar2
    :
    if nVar1=1
        ; I need new registers just for this section of code
        defbegin
            defi nLoc1
            defp pLoc2
            ; use these variables just in here
            cpy -1, nLoc1
            :
        defend
```

```

        ; defend will make sure that
        ; the registers allocated to nLoc1 and
        ; pLoc2 are now made available
        ;for use in the rest of the program.
    endif
    :
    defend

```

Tracing

Several tracing facilities can help you debug VP programs.

Trace device

The trace device is used for troubleshooting application programs. It is typically used for displaying debugging information and error messages. The **TRACEF** macro writes messages to the trace device.

×

By default the trace device is the screen but can be redefined to be a file using the shell command `ftrace <filename>`. Thus, `ftrace myfile.txt` would cause trace output to go to the file `myfile.txt` rather than the screen. Type `ftrace` to set the trace device back to the screen again.

Tracef

This macro enables messages and data to be written to the trace device. The syntax matches `printf`'s. For instance VP source can contain a statement like:

```

    tracef "error code : %d\n",err_code

```

tracef writes characters directly to the trace device, without buffering. The **tracef** macro has code that waits for an acknowledgement from the trace device to say that the characters have been printed.

Note that this behaviour of **tracef** is blocking by its nature, unlike **printf** where characters are printed to a buffer that is flushed only when the buffer is full or when the buffer is specifically flushed by a library function.

By contrast, **printf** is non-blocking, and is thus not ideal for debug purposes as the next instruction after a `printf` may cause the program to crash, while the characters that **printf** was to print may not yet have been flushed from the buffer, making tracing of execution with **printf** very difficult.

tracef was designed to circumvent this problem, much as **kprintf** was in the Classic Amiga ROM kernel. Both **printf** and **tracef** will work in a multi-processor environment where the trace device, for **ftrace**, or **stdout**, for **printf**, are on a different processor to the one executing the trace statement.

ktrace.log

This is the trace device for the kernel developer. If you are developing low-level code and your system should crash at any point, close examination of **ktrace.log**, which can be found in the root directory, may provide useful information, given luck and a following wind.

Using pre-defined macros

VP programmers can develop macros to aid productivity as well as readability of code. It has also been indicated that many of the programming structures, for instance, while endwhile are in fact implemented as macros. There are also a large number of predefined macros to assist in common programming tasks like:

- error handling
- linked list manipulation
- general purposes, such as multi-byte to wide characters

Error checking macros

Pre-defined error checking macros include **boolerrno**, **boolnoterrno**, **breakiferrno**, **iferrno** and **errorf**.

The first four are examples of macros that test for error indications. These macros can be used in conjunction with routines that return an **errno**, a value in the range -128 to -1, when some error condition has been identified within that routine.

Consider a function that configures a small text buffer. It will return a pointer to that buffer, assuming that memory could be allocated and internal initialisation could be completed. If the routine is successful it returns a valid pointer, but if not it returns a value in the range -1 to -128 in the register, say **p0**. The error checking code would be as follows:

```
qcall demo/example/makebuff, (i0:p0)  
boolerrno p0, buff_fail
```

boolerrno checks the value in **p0**, and if it is in the range -1 to -128 it causes a jump to the label **buff_fail**. As can be seen from the preceeding list there are variations on this basic macro.

Another useful macro is **errorf**. This macro allows a formatted output of data along the lines of **printf** or **tracef**. However, its output is to the file **error.log**, which is located in the root of the Amiga directory structure. It is useful for logging error codes and general post-mortem debugging.

```
iferrno i0  
    errorf "Error %d occured.\n", i0  
endif
```

Linked list macros

One common programming task is to manipulate a number of objects in a linked list data structure. Amiga has macros ready-coded to reduce the effort required to handle linked lists.

<i>initlist</i>	initialises a linked list
<i>addhead</i>	add a node to the head of a list
<i>addtail</i>	add a node to the tail of a list
<i>addnode</i>	add node after node specified
<i>addnodeb</i>	add node before node specified
<i>succ</i>	get next node pointer
<i>succnode</i>	get next node pointer, jumping to label if at end of list
<i>pred</i>	get pointer to previous node
<i>prednode</i>	get pointer to previous node, jumping to label if at head
<i>remhead</i>	remove node at head of list; jump to label if list empty
<i>remove</i>	remove the specified node from a list

These are close analogues of the Classic Amiga EXEC list functions, but the new system has more, and macros implement them more efficiently on modern pipelined processors. Some of the names correspond exactly, but the functionality is not quite the same. Classic Amiga's **RemHead** function returns a null value if the list is empty, whereas the new macro takes an extra parameter and uses that as a label for program continuation in the corresponding circumstance.

In order to use these macros the programmer has to understand the data structures that they manipulate. There are three main structures:

List header	defined in equs.inc
List node	defined in equs.inc

User List node user-defined

The list header is defined exactly like a `MinList` in the Classic Amiga system:

```
structure
    pointer LH_HEAD
    pointer LH_TAIL
    pointer LH_TAILPRED
    size LH_SIZE
```

The list node is defined reassuringly like a Classic Amiga `MinNode`:

```
structure
    pointer LN_SUCC
    pointer LN_PRED
    size LN_SIZE
```

The user defined node is application-specific but an example is:

```
structure
    struct NDHDR, LN_SIZE ; name, size
    int32 APPDATA
    size ND_SIZE
```

The basic list node structure has been embedded in the user node structure to facilitate list manipulation.

Before using the list macros, it is necessary to allocate memory for the list head:

```
cpy LH_SIZE, i0
qcall lib/malloc, (i0:p0) ; insert NULL error check code
```

We can then initialise the list with `initlist p0`. This provides us with an empty list. To add items we allocate memory for the nodes to add and then use the appropriate macro, for example:

```
cpy ND_SIZE, i0
qcall lib/malloc, (i0:p1)
; insert NULL error checking code
cpy 1234, [p1+APPDATA]
```

At this point the node has been created and initialised. This adds it to the list:

```
addhead p0, p1, p2
```


p0 points to the list, p1 points at the node to add and p2 is a scratch register, used internally by the macro. The macros to remove nodes are used in the same way.

The macros **succ**, **succnode**, **pred** and **prednode** are provided to walk along the list. The **succnode** macro takes three parameters: a pointer register to receive the pointer to the next node in the list, a pointer register that points to the current node and finally a label. The label provides a useful exception handling facility as succnode causes execution to jump to this label should the current node be at the tail of the list, in which case there is no 'next node'. For example:

```
succnode p2,p1,tail_list
:
tail_list :
printf "The end is nigh - you have run out of list.\n"
go exit
```

The current node is in p1 and the next node comes back in p2. If p1 points to the current tail of the list then a jump to tail_list occurs.

General purpose macros

General purpose macros like mbtowc and fprintf carry out tasks like conversion from multi-byte to wide characters or input/output. Further details are in the *VP Reference Manual* on your developer CD.

Using Libraries

VP programmers have at their disposal a powerful array of built in library functions. Currently Amiga has a complete library of ANSI C functions, as well as a significant part of the POSIX specification. The library functions are called using a qcall as described earlier. The following code demonstrates the use of some POSIX directory functions:

```
cpy.p dirname,p0
qcall lib/opendir,(p0:p1)
if.p p1=NULL
printf "Failed to open %s\n",dirname
go exit
endif
loop
qcall lib/readdir,(p1:p2)
if.p p2=NULL
```

```

        printf "End of listing.\n"
        break
    endif
    add DE_NAME,p2
    printf "-- %s\n",p2
endloop
qcall lib/closedir,(p1:i0)

```

Library functions are implemented as tools in VP and called like user defined tools. Library tools are developed in VP for optimal portability and performance.

Native Processor Coding

Nine times out of ten the advantages of programming in VP outweigh the slight disadvantage that its idea of optimal code might not always match your own, if you know the specific details of a system. But every so often - albeit less often than some hackers would like to think - you may need to wring the last ounce of performance out of a certain system.

Amiga is equal to the challenge, because it allows you to write code directly for your native processor, and integrate that with friendly, portable, reliable VP. And if the processor turns out not to be the one you expect, everything can still work, though perhaps less optimally.

The information in this section is not normally needed by programmers writing only for the Amiga Virtual Machine. It is intended for use by programmers writing system-specific hardware device drivers or optimised native code sections, or unworldly things like benchmarks and demos. We try to cater for all tastes!

Changes to an Application Source File

In order to be able to write a tool in native code a few simple changes are required. The following example illustrates writing in PPC assembler. The same principle holds for any other supported processor architecture. Code starts as usual:

```
.include 'taort'
```

Changing the tool definition

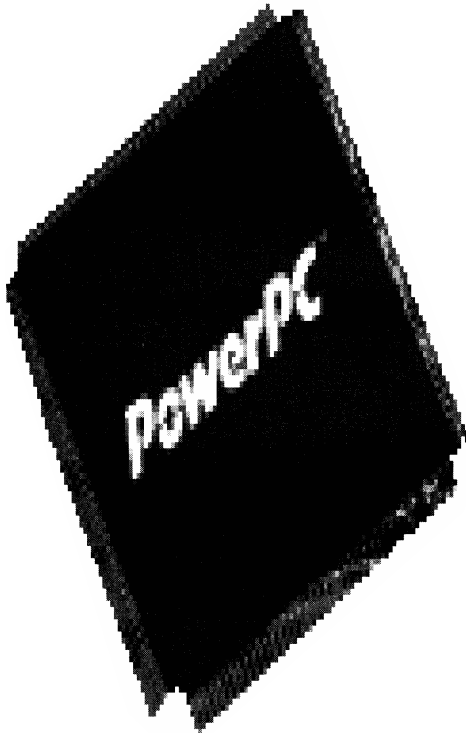
The language in which the tool is going to be written must be specified, after the tool macro and the tool name, in place of VP.

```
tool 'demo/example/hello',PPC_B
```

No further changes are required and the rest of the tool is coded in the language for the specific processor. When assembled, a tool file is created with a suffix unique to the target processor.

VP or native Tool Selection

At load time Amiga's unique dynamic binding technology searches the tree of tool dependencies. Amiga selects all the tools defined, only taking VP (00) tools if a tool with the same processor number as the target processor is not available. It is therefore strongly recommended that an equivalent VP tool is created for each native tool, to ensure portability.



- 00 VP
- 01 M-CORE
- 02 PSC-1000
- 03 i86 common
- 04 i386
- 05 SH-4
- 06 Thumb
- 07 DP-1000
- 08 Arm6
- 09 Arm6f
- 10 SA110
- 11 R4000_L
- 12 R4000_B
- 13 I386f
- 14 I486
- 15 Pentium
- 16 PPC_B
- 19 PPC_L
- 20 CW400x
- 21 R4100_L
- 22 R4100_B
- 23 PentMMX
- 24 Pent II
- 25 V850
- 26 CF52xx
- 27 V850e

VPcode support for the Power PC provides a bridge from the Classic Amiga to the new systems. A couple of gaps in the above list have been allocated for processors that we're not allowed to tell you about yet.

This list is subject to regular alterations as new processors are developed and integrated into the Amiga repertoire. The beauty of VP is you don't even need to know what they are. L and B suffixes denote little and big-endian configurations.

New processors can be implemented with a VP translator and your programs will run efficiently on them, without any need for you to recompile or check the code. After the trials of converting the Classic Amiga from 68K to PPC - and Apple's painful experience making the same transition - this has to be good news. If you really need hand-optimised code, you can add a processor-specific tool which will get used instead of the machine-generated one if the hardware is appropriate.

Special Registers

The Virtual Processor has a few registers that defy easy categorisation. This section discusses them, and their uses. It's sometimes useful to be able to read these, but never a good idea to alter their values because the system does not expect them to change unless it is the one doing the changing.

The Stack Pointer, SP

The stack pointer points to the lowest address containing valid data. If the programmer wishes to manipulate the stack directly, it should be noted that it is a downward growing stack. The newest entries use the lowest addresses. Classic Amiga systems use the same convention for their user and supervisor stacks, although gravity mitigates against this arrangement in the real world!

The VP stack is always aligned to a CPU-specific boundary and this alignment is automatically preserved whenever SP is adjusted. 68000-compatibility meant that Classic Amiga stacks are 16 bit word-aligned, and suffered a performance hit when long words were stacked at addresses that are not evenly divisible by four.

The Parameter Pointer, PP

The parameter pointer is a special read only register, which is set by routine entry code to the stack pointer of the caller before the call occurred. It can be used for passing parameters via the stack. The standard parameter passing convention uses registers, not stack, in the vast majority of cases. Usually PP is not implemented as a physical register - rather, it is a known offset from SP. It follows that the value of PP equals the value of SP immediately before a subroutine call.

PP demonstration

This example snippet shows how PP is used in a real VP program:

```
structure
int32 param1
int32 param2
int32 param3
size ppdemo_size
allocstruct ppdemo_size, p0
cpy 1,[p0+param1]
cpy 2,[p0+param2]
cpy 3,[p0+param3]
gos ppdemo, (-:i0)
tracef "result = %d",i0
freestruct ppdemo_size
ppdemo:
ent -:i0
cpy [pp+param1],i1
cpy [pp+param2],i2
cpy [pp+param3],i0
add i1,i2
mul i2,i0
ret
```

The Link Pointer, LP

The link pointer contains the return address which the routine should jump to after tidying up the stack in a **ret**. This register may be read and written, although changing it to point elsewhere will cause **ret** to return to a different location...

The Global Pointer, GP

The global pointer points to data that is thread wide. GP points to an area of thread-wide memory, which can be both global and static data. The size of the memory block is specified in the main tool. The global pointer should not be altered by the programmer.

The Amiga Kernel

Kernel Overview

The Amiga kernel is an advanced, portable real-time kernel written in VPcode. It uses services provided by the **CPU Isolation Interface (CII)**, and **Platform Isolation Interface (PII)** and device drivers such as the timer device, in order to achieve true platform independence.

This chapter provides a brief overview of the Kernel's features. For more detailed information please refer to the *Kernel Reference Manual* which we provide on your development CD in PDF format.

Kernel Features

These are some of the cool features of the Amiga kernel:

- portable
- small
- fast
- various real-time scheduling algorithms available as standard
- user defined scheduling algorithms
- pre-emptive multitasking
- wide range of Inter Process Communication (IPC) mechanisms
- task priorities
- customisable
- dynamic binding
- heterogeneous symmetric multi-processing

Kernel Services

The kernel provides these main services:

- tool management
- process management
- inter-process communication
- memory management
- signals
- timers
- callbacks

- scheduling
- atoms
- named data areas (NDAs)
- entropy collection services

The following sections describe each of these areas in more detail.

Tool Management

The kernel has facilities for loading and binding tools. The kernel uses a special device driver called the **tool loader** to search for tools on disk and load them. The tool loader calls translators if the tool contains VP byte codes, and the translator returns the tool in native form for the current processor.

Typically a tool is loaded when a process qcalls the specified tool. The actual mechanism depends on the particular type of qcall made. There are three types of qcall: non-virtual, virtual and virtual+fixup.

Non-virtual Qcalls

The default type is non-virtual. With this qcall all tools referenced by the calling tool are loaded, and translated if required, when the calling tool is loaded. Thus all the tools that may be required are available in memory at the start of program execution.

This mechanism provides maximum run-time speed, but is less efficient in its use of memory than the alternatives because tools that have been loaded may not actually be required, depending on the sequence of execution within the program.

Virtual Qcalls

With virtual qcalls the referenced tool is not loaded until dictated by the run-time execution path. At this point the tool is searched for in memory and if not present the tool is loaded from disk.

On completion of execution of the virtual tool the calling process decrements the tool's reference count. The reference count is a count of all processes currently referencing (using) that tool.

Only one copy of a given tool exists in memory at any one time, so if four processes were using a particular tool simultaneously it would have a reference

count of four and there would not be four copies of the tool in memory. When the reference count of a tool is zero it may be flushed from memory by the kernel.

The advantage of the virtual tool is that memory requirements when the calling tool is loaded are minimal, as virtual tools only get loaded when required. The disadvantage is that there is a small run-time performance hit as the requested tool is loaded and translated.

Virtual+fixup Qcalls

The virtual+fixup tool is similar to the virtual tool in that it is only loaded on request, if not already in memory. However, when the virtual+fixup tool completes, the reference count of the tool is not decremented. This means that the virtual+fixup tool will not be flushed should the kernel decide to flush dereferenced tools when resources are low.

The advantage of this is that any subsequent calls to the virtual+fixup tool will not require loading and translation of the tool. So on the first call to the virtual+fixup tool, performance is comparable to a virtual qcall and for subsequent calls performance is comparable to a non-virtual qcall.

These calls patch the original call in the calling tool, which is why subsequent calls are almost as fast as non-virtual. Virtual calls cannot do this, so even if the tool is already in memory it has to be found on the tool list, and is therefore slower.

Tool management calls

The kernel provides the following tool management calls. Please refer to ‘The Kernel Reference Manual’ for more information. Note that these tools will only be used under special circumstances; the more usual approach is to load and manage tools via the qcall mechanism.

`sys/kn/tool/add` - adds translated tool to tool list. The tool then needs to be opened

`sys/kn/tool/open` - scans memory for a tool; if not found the tool loader is called

`sys/kn/tool/deref` - decrement reference count

`sys/kn/tool/ref` - increment reference count

`sys/kn/tool/flush` - flush all unreferenced tools from memory

`sys/kn/tool/flushname` - flush a specified unreferenced tool from memory

`sys/kn/tool/lookup` - returns pointer to the header of a tool containing this address

`sys/kn/tool/getname` - given pointer to the header of a tool, returns name in buffer

Process Management

A process can be thought of as a program in execution. The Amiga kernel supports multiple processes running in parallel. True parallel processing is supported through symmetric multi-processing on multiple processors, which can be dissimilar, as well as the more traditional multi-processing through timeslicing.

On many operating systems there is a distinction between threads and processes. On these systems threads have the following properties:

- It's usually faster and more memory efficient to create threads than processes
- threads can share certain types of data such as file descriptors or signal tables
- threads spawned from an application usually operate within its memory space, allowing them to share data more efficiently and without memory protection

Amiga programmers do not explicitly create threads; rather the programmer spawns new, lightweight processes. Amiga processes have these properties:

- the creation of processes in Amiga is fast and memory efficient
- memory protection between processes is not enforced
- the sharing of system resources and data between processes is supported for instance they have a common file table, and so on.

In the Amiga system a thread is thought of as a route through executable code in a sequence of tools.

An Amiga process can have any of the following states:

- Non-existent
- Dormant
- Suspended
- Sleeping
- Sleeping and suspended
- Ready
- Running

The state transitions are documented more fully in *The Kernel Reference Manual*. However, for ease of reference, the kernel process management calls and the resultant state transitions are tabulated below. Please note that `sys/kn/proc/*` calls fall into one of two categories, depending on whether they act on the calling process itself or act on another specified process.

In the following table below calls that operate on the calling process, so that the calls acts on itself, are marked with a '+' sign. Calls that do not change the state of the process they address are marked with a '*' sign after their description.

Name	Description	Initial State	Final State
sys/kn/proc/create	Create a new process	NON-EXISTENT	DORMANT
sys/kn/proc/delete	Destroy a process	DORMANT	NON-EXISTENT
sys/kn/proc/start	Start a process	DORMANT	READY
sys/kn/proc/exit +	Terminate calling process	RUN	DORMANT
sys/kn/proc/terminate	Terminate specified process	READY	DORMANT
		SLEEP	DORMANT
		SUSPEND	DORMANT
		SLEEP&SUSPEND	DORMANT
sys/kn/proc/sleep +	Block the calling process	RUN	SLEEPING
sys/kn/proc/wake	Cancel a self-imposed sleep	SLEEP	READY
		SLEEP&SUSPEND	SUSPENDED
sys/kn/proc/suspend	Block the specified process	READY	SUSPENDED
		SLEEP	SLEEP&SUSPEND
		SUSPEND	SUSPENDED
		SLEEP&SUSPEND	SLEEP&SUSPEND
sys/kn/proc/resume	Unblock specified process	SUSPEND	READY
		SLEEP&SUSPEND	SLEEPING
sys/kn/proc/deschedule +	Give up calling process's current timeslice	RUN	READY
sys/kn/proc/setparams	Modify the scheduling parameters of specified process		*
sys/kn/proc/getparams	Return the scheduling parameters of specified process		*
sys/kn/proc/disable_sched	Stop the scheduler from running		*
sys/kn/proc/enable_sched	Allow the scheduler to run		*
sys/kn/proc/wait	Wait for a child process to terminate, and delete it	DORMANT	NON-EXISTENT
sys/kn/proc/chld	Wait for a child process to terminate		*
sys/kn/proc/chpri	Change the priority of the calling process		*
sys/kn/proc/chppid	Change the parent process of a process		*

Process creation and deletion

Here are brief descriptions of the process-related kernel services:

sys/kn/proc/create - create a new process (which must then be started)
sys/kn/proc/delete - delete a process
sys/kn/proc/start - start a created process

sys/kn/proc/exit - put calling process (itself) in dormant state
sys/kn/proc/terminate - terminate specified process

Process creation helper functions

sys/kn/proc/exec/local - create and start a process on the same processor
as the calling process
sys/kn/proc/exec/remote - create and start a process on the specified processor
sys/kn/proc/exec/any - create and start a process on a processor chosen by
the kernel

Process control

sys/kn/proc/sleep - put calling process to sleep (self block)
sys/kn/proc/wake - wake specified process
sys/kn/proc/suspend - suspend specified process
sys/kn/proc/resume - unsuspend specified process
sys/kn/proc/wait - Wait for a child process to stop or terminate and delete it
sys/kn/proc/chld - wait for a child process to terminate
sys/kn/proc/chpri - change the priority of a calling process
sys/kn/proc/chppid - set the parent process PID to a new value

Process scheduling

sys/kn/proc/deschedule - give up current time slice
sys/kn/proc/disable_sched - stop scheduler
sys/kn/proc/enable_sched - start scheduler

Process parameters

sys/kn/proc/setparams - modify scheduling parameters of another process
sys/kn/proc/getparams - get scheduling parameters of another process

Spawning

sys/kn/proc/spawn/make - creates a spawn structure for sys/kn/proc/create
sys/kn/proc/spawn/emake - POSIX style version of sys/kn/proc/spawn/make
sys/kn/proc/spawn/modfd - modify file descriptor in spawn structure
sys/kn/proc/spawn/modglobs - add global data initialisation in spawn structure

Scheduling

The kernel provides the following scheduling algorithms :

- Round Robin
- Rate Monotonic (RM)
- Minimum Laxity First (MLF)
- Maximum Urgency First (MUF)
- Earliest Deadline First (EDF)
- Deadline Monotonic Scheduling (DMS)

In addition to these, user defined scheduling algorithms are also supported. It is also possible to utilise several scheduling policies at the same time by assigning different policies to different task priority ranges.

A process can have a priority in the range 0 to 255, where 0 is the highest priority.

The PID or Process ID

Amiga defines a portable interface to the process table data structure. This table can be fixed size or hierarchical depending on the Amiga's configuration.

The fixed size model is suitable for small, embedded systems, where the number of processes is limited and known at the outset. The hierarchical process table is more suitable for systems where processes are dynamically downloaded, where there are a large number of processes or where the actual number of processes is otherwise unpredictable. Note that customer definable process tables are made possible by implementing the process table access tools in the `sys/kn/proc/pid` directory. The default Amiga process table is the hierarchical process table.

Amiga supports both local and network Process IDs. Normally local PIDs are used, but where a number of Amiga kernels are on a network and need their processes to communicate network unique PIDs must be assigned to processes. The kernels agree on which unique numbers can be allocated so that a conflict in network PIDs does not occur.

The programmer should be aware that it is possible to spawn remote processes from a process that has a local PID. However, signals from the child process may not pass back to the parent process, as there is no network PID for it. See *The Kernel Reference Manual* for more information on this and other PID issues.

Inter Process Communication (IPC)

The Amiga kernel supports several communication mechanisms. These include:

- Counting semaphores
- Mutexes
- Event Flags
- Mailboxes

Counting Semaphores

These are the Kernel Semaphore operations:

sys/kn/sem/init - initialise a semaphore
sys/kn/sem/destroy - destroy an unnamed semaphore
sys/kn/sem/trywait - wait on a semaphore, non-blocking
sys/kn/sem/wait - wait on a semaphore
sys/kn/sem/timedwait - wait on a semaphore with timeout
sys/kn/sem/post - post to a semaphore
sys/kn/sem/getvalue - get the value of a semaphore
sys/kn/int/sem/post - post to a semaphore from an interrupt

Mutexes

This is a list of the Mutex operations, which implement a concept similar to the locks in Classic AmigaOS:

sys/kn/mtx/init - initialise mutex
sys/kn/mtx/destroy - destroy mutex
sys/kn/mtx/lock - lock mutex
sys/kn/mtx/trylock - lock mutex, non-blocking
sys/kn/mtx/timedlock - lock mutex with timeout
sys/kn/mtx/unlock - unlock mutex
sys/kn/mtx/islocked - return lock status of mutex

Event Flags

These are the event flag operations supported by the Amiga Kernel:

- `sys/kn/evf/init` - initialise an event flag structure
- `sys/kn/evf/destroy` - destroy an event flag
- `sys/kn/evf/set` - set the flag pattern of an event flag
- `sys/kn/evf/clr` - clear the flag pattern of an event flag
- `sys/kn/evf/wait` - wait on event flag until specific condition occurs
- `sys/kn/evf/trywait` - test event flag for specified event flag pattern, non-blocking
- `sys/kn/evf/timedwait` - wait on event flag for specific condition, with timeout
- `sys/kn/evf/info` - get event flag information

Mailboxes

Mailbox is the new name for what Classic Amiga programmers know as a message port. These are the mailbox operations:

- `sys/kn/mbox/alloc` - allocate a mailbox
- `sys/kn/mbox/free` - free a mailbox
- `sys/kn/mbox/send` - send message to mailbox
- `sys/kn/mbox/read` - read mail from mailbox
- `sys/kn/mbox/tryread` - read mail from mailbox, non-blocking
- `sys/kn/mbox/timedread` - read mail from a mailbox, with blocking and timeout

Synchronisation Groups

Extensive facilities are available for collecting synchronisation objects together into a collective structure referred to as a **synchronisation group**. Semaphores, mutexes, event flags and mailboxes can all be added to a specific group. This makes it easier to write real-world applications that need to wait for one of several events to occur.

Memory Management

The Amiga system uses an object based memory allocation scheme.

Memory objects

All these memory objects are available:

- default application data memory object
- default stack memory object
- system data memory object
- mail message memory object
- code memory object

These memory objects could each be mapped to different allocator classes, or all mapped to a single allocator class, depending on the system and its configuration.

Allocator classes

The following allocator classes are available:

`sys/kn/mem/std` - generic allocator

`sys/kn/mem/debug/std` - debug version of the generic allocator

`sys/kn/mem/nstd` - faster and deterministic version with slightly higher overhead

`sys/kn/mem/pii` - an allocator that calls the underlying host OS to allocate and free

`sys/kn/mem/buddy` - uses buddy block algorithm, deterministic allocation and free

The details of underlying memory objects and allocator objects are hidden from the application programmer. The programmer uses the following tools to allocate and free memory:

`sys/kn/mem/allocstk` - allocate memory for process stacks

`sys/kn/mem/allocdata` - allocate memory for data

`sys/kn/mem/allocsys` - allocate data memory for the system itself

`sys/kn/mem/alloccode` - allocate memory to store dynamically loaded tools

`sys/kn/mem/allocmail` - allocate memory to be sent as mail messages

`sys/kn/mem/allocdef` - used by libraries to allocate data memory which needs to be shared by a number of processes and which must exist after the allocating process terminates

Crucially unlike the Classic Amiga, space for messages and other communication between processes must be specially allocated. You can't just build the equivalent of a message port and hope that the process will be able to see it. It might not even be on the same processor.

The new system uses a special pool for mailboxes and other shared structures, ensuring that they are not obscured by memory management, appear in the shared space on multiprocessor systems with a unified memory architecture, or get copied if the other processor is on a remote network.

This means that communication can be as efficient as it would be on an unprotected Classic Amiga, or as flexible as it could be on a system like QNX, where messages are invariably copied. These calls ensure that we can get the best of both worlds, and allow memory protection and virtual memory to be added to the Amiga system without the problems caused by the MEMF_PUBLIC scheme on old Amigas.

All of the memory allocator tools share the same interface, as follows:

```
sys/kn/mem/alloc<type>(i0:p0 i0)
```

Inputs:

i0 = number of bytes requested

Outputs:

p0 = pointer to allocated block, or NULL if allocation failed

i0 = number of bytes allocated (maybe greater than that requested)

They call on an intermediate memory allocation routine `sys/kn/mem/alloc` which takes a pointer to the appropriate memory object.

Other useful tools

`sys/kn/mem/free` - frees up specified memory block

`sys/kn/mem/realloc` - reallocate a specified memory block

`sys/kn/mem/check` - checks structure of all system memory objects

`sys/kn/mem/lookup` - get pointer to named memory object

Timer Management

Amiga provides two types of timer - periodic and monoshot. A periodic timer provides a succession of responses; the monoshot is a one-off facility.

When the time comes, the timer can wake a specified process, send a signal to a specified process, or call a timer handler. The timer functions include these:

`sys/kn/timer/set` - set up a timer using the priority of the calling process

`sys/kn/timer/dset` - set up a timer using priority specified in timer data structure

`sys/kn/timer/unset` - unset timer

Interrupt Handling

There are special rules for interrupt handlers, because they may run at almost any time, whatever the rest of the system is doing. As on the Classic Amiga, the set of routines which can be called within an interrupt handler is restricted. If you need to do more than these calls allow you must communicate with a process to do the work for you, when the system is sure to be ready.

The following restrictions apply to interrupt handlers:

- Stack - interrupt handlers execute with a stack setup by the PII. This stack is of fixed size. Overrunning the stack will cause a system crash.
- Register usage - interrupt handlers should preserve all native registers. Use `entih` to ensure this.
- Memory usage - memory accessed by the interrupt handler should be locked using `sys/kn/mem/lock`. This applies to both code and data memory.
- Code usage - The following tools may be called from an interrupt handler:

`sys/kn/int/evf/set`

`sys/kn/int/mbox/send`

`sys/kn/int/proc/wake`

`sys/kn/int/proc/suspend`

`sys/kn/int/proc/terminate`

`sys/kn/int/proc/setparams`

`sys/kn/int/proc/getparams`

`sys/kn/int/proc/resume`

`sys/kn/int/sem/post`

`sys/kn/int/sig/kill`

Signals

POSIX 1 signals are supported but POSIX 4 signals are not currently supported.

Signals are generated by these categories of events:

- Hardware exceptions - illegal instructions, invalid memory references, arithmetic exceptions and other processor-detected problems.
- Abnormal software conditions - writing to a pipe with no readers, termination of a child process, attempts to virtually call tools that do not exist, and so on.
- User initiated events - such as a user pressing ctrl-C or ctrl-Y at the keyboard, or running the **kill** program.
- Program initiated events - alarm expiration, IPC using SIGUSRn signals, etc.

Signal actions

There are three possible actions for a signal. SIG_DFL performs the default action for signal. The SIG_IGN action means ignore the signal. Otherwise they use a pointer to a signal handling function.

On delivery of the signal the receiving process calls the signal handler indicated by a pointer. The handler will normally return to the receiving process, but may be made to return to a location specified by `setjmp` by using the `longjmp` tool.

Many systems use the concept of a 'supervisor' or 'privileged mode' and a 'user mode'. In these systems signals can only be delivered to a process while it is in user mode. The new Amiga does not make such distinctions.

Modes give protection from signal delivery while sensitive operations such as I/O or manipulation of system data structures are being performed. Although the new Amiga has no concept of supervisor mode or user mode, it can provide the same level of protection as supervisor mode by switching off signal delivery to a process under certain circumstances.

This does not affect a process's signal mask or any pending signals. Most critical regions within the system are protected by mutexes, which provide the MTX_SIGMASK flag. If this flag is set, signals will be switched off automatically by the mutex code when a process becomes the owner of a mutex. Signals are

effectively re-enabled when the process unlocks the mutex. This removes the danger that, for example, a signal handler may `longjmp()` out of a critical region. The `MTX_SIGMASK` feature is of particular interest to programmers of software that would normally be thought of as running in supervisor mode, such as Linux or Qdos device drivers.

Signal functions

These are some of the signal functions:

- `sys/kn/sig/kill` - send a signal to a process
- `sys/kn/sig/action` - examine or change signal action
- `sys/kn/sig/procmask` - examine or change blocked signals
- `sys/kn/sig/pending` - examine pending signals
- `sys/kn/sig/suspend` - wait for a signal

Sets of Signals

These functions are handy for manipulating sets of signals:

- `sys/kn/sig/emptyset` - creates an empty set
- `sys/kn/sig/fillset` - creates a full set
- `sys/kn/sig/addset` - adds a signal to a set
- `sys/kn/sig/delset` - deletes a signal from the set
- `sys/kn/sig/ismember` - tests to see if signal is a member
- `sys/kn/sig/setflag` - disable or enable signal handling

Callbacks

A callback is a function or tool that can be called from another process, but operates in the context of the process that 'owns' the callback. An example of the use of callbacks is in asynchronous device drivers. When a lengthy I/O task completes a callback to a handler function can be invoked.

This can then indicate that the operation is complete and any further required operations can now be carried out. This prevents the device driver from having to enter a wasteful polling loop or sleeping and thus blocking the process that calls the driver.

Callback facilities include these:

sys/kn/callback/set - post a callback
sys/kn/callback/unset - unset a callback
sys/kn/callback/setflag - disable/enable callbacks
sys/kn/callback/process - process any pending callbacks
sys/kn/callback/occurred - returns value of PF_CALLBACK_OCCURRED flag
sys/kn/callback/clr_occurred - clears the PF_CALLBACK_OCCURRED flag

The Kernel Reference Manual contains much more information about callbacks.

Named Data Areas (NDAs)

The basic concept behind the NDA is that the Amiga kernel allows a pointer to a data area to be associated with a string or 'name'. This has the advantage that a number of processes running on a chip can access a data area without having to know a specific pointer. This is very useful for implementing processor wide data structures that may be shared by a number of processes.

The NDA services are:

sys/kn/nda/name - associate the specified pointer with the specified string
sys/kn/nda/del - delete the NDA record for the specified string
sys/kn/nda/find - lookup the NDA record for the specified string

Atoms

Atoms allow character strings to be represented more efficiently as a unique integer. The kernel provides facilities for creating, deleting and looking up atoms. Each processor in an Amiga system possesses an atom table, which records the mappings between a string and its associated integer.

The Amiga kernel stores atoms in different ways, depending on whether they are created statically when the system is generated, or dynamically by the kernel.

Static atoms

Static atoms last for the lifetime of a system and can be in ROM, so that they cannot be modified. They have no reference count and do not cease to exist if unused by the system. Static atoms need less memory than dynamic atoms and can be compressed to a greater degree without a large run-time penalty.

Static atoms are stored in one table, while the data relating to their associated strings is held in another compressed data table. The atom table is pointed to by `KN_ATOMTABLE` and the text data table is pointed to by `KN_ATOMTEXT`.

Dynamic atoms

Dynamic atoms are stored in a singly linked list, pointed to by the kernel variable `KN_ATOMLIST`.

Amiga atom functions include these:

`sys/kn/atom/add` - return atom value for specified string; creates atom if required
`sys/kn/atom/del` - dereferences the specified atom, deleting it if unreferenced
`sys/kn/atom/find` - return the atom value corresponding to the specified string
`sys/kn/atom/getname` - return a copy of string corresponding to the specified atom

Atomic Linked List Functions

Linked list operations are sensitive to being interrupted in a multitasking environment. It is possible for a list to be left in an inconsistent state. This could easily cause a system crash where the lists manage important system information, for example, memory allocated and free lists. For this reason the kernel provides atomic list manipulation functions.

The functions provided are:

`sys/kn/atomic/addhead` - add a node to the head of the list
`sys/kn/atomic/addtail` - add a node to the tail of the list
`sys/kn/atomic/addnode` - add a node after the specified list node
`sys/kn/atomic/addnodeb` - add a node before the specified list node
`sys/kn/atomic/removehead` - remove the node from the head of the specified list
`sys/kn/atomic/removenode` - remove the specified node from its list
`sys/kn/atomic/movelist` - move the entire contents of one list onto another

Kernel Device Functions

Amiga devices are made available to applications through the use of a mount table. Each mount table record contains the device ID, the name of the mount point and some flags.

The device ID is a 64-bit value, divided into two parts. The least significant 32-bits hold the device instance pointer. The most significant 32-bits identify the processor on which the device is situated.

The use of the **devstart** command is discussed in the device driver section of this manual.

The flags are specified at the time the device is mounted. These can be used to specify whether the device is network visible, or only visible to processes on the same processor.

These are some of the main device functions:

sys/kn/dev/lookup - look up a device in the system mount table

sys/kn/dev/rlookup - look up a device in the system mount table

sys/kn/dev/mount - add a device to the system mount table

sys/kn/dev/unmount - remove the specified device from the system mount table

sys/kn/dev/mount_delayed - adds a delayed-mount record for a device into the system mount table

The lookup function takes a name and returns a device, while rlookup (R for 'reverse') takes a device pointer and returns the name of that device. The difference between MOUNT and MOUNT_DELAYED corresponds to states of the Classic Amiga mountlist or tooltype option MOUNT=1 or MOUNT=0, respectively.

Kernel Entropy Collector

The purpose of the entropy collector is to provide a source of high quality randomness. This randomness can then be used for applications that require randomness, such as key generation utilities.

The entropy in the system is ‘trapped’ by the device drivers in the system and then pooled by the kernel into an entropy reservoir. Requests for entropy from applications can only be serviced by the kernel if there is sufficient entropy in the entropy reservoir or entropy data pool.

In cases where entropy is not required or is not supported by underlying hardware, the kernel entropy collector can be replaced by a null version. At system build time it can be specified whether the real entropy collector is to be used or not. If not a group of stub routines are used. These support the device driver entropy interface but actually perform null operations.

Entropy collector functions include:

- sys/kn/entropy/add - add entropy to the collector data pool
- sys/kn/entropy/add_time - add timer entropy to the pool
- sys/kn/entropy/reg_time - register the timer clock resolution
- sys/kn/entropy/get - extract entropy from collector pool
- sys/kn/entropy/getrand - extract entropy from collector pool
- sys/kn/entropy/get_async - extract entropy asynchronously
- sys/kn/entropy/get_abort - cancel asynchronous entropy request

Kernel Time Functions

- sys/kn/timer/set - sets up a timer
- sys/kn/time/get - get the kernel time

Ebug

Ebug is the Linux hosted debugger. In order to use ebug it is necessary to make sure that you are using an Amiga system built with the checking translator.

When debugging it is important to remember to use the `-g` option with the assembler, as this adds debug information to the tool. This in turn will allow you to disassemble the tool with the `dis -s` option, which shows the source that goes with the corresponding VP byte codes. For example:

```
$asm -g demo/example/hello
```

```
$dis -s demo/example/hello.00
```

When assembling the `.00` extension is optional, but when disassembling the `.00` extension is required, as a tool may be implemented with several extensions, such as the generic `hello.00` and `hello.16` for a PPC. The disassembler can be used to disassemble tools with any supported extension:

```
$dis -s demo/example/hello.15
```

ebug is particularly useful for identifying common problems such as:

- insufficient stack space (stack overflow)
- incorrect parameter passing
- misaligned memory access
- incorrect use of `noret`

When these problems occur ebug will give clear information about the nature of the problem and where it occurs in the code. To see an example of this look at `demo/example/debug.asm` in the Amiga build. This would also be a useful program to use to get familiar with the operation of ebug.

As ebug is useful at trapping common types of bugs it is recommended that all development work be carried out via ebug and the checking translator, where possible. The checking translator results in tools that are much larger than if the normal translator is used, as a result of the extra debug code that is inserted into the tool.

Ebug will also return useful information in the event of a program crash. Ebug shows the last tool loaded into memory before the crash occurred, which can be helpful in tracking a bug. Also ebug will generate messages showing which tools

are being loaded and unloaded. To demonstrate this start an ebug session, at which point ebug and an Amiga box should load. When in ebug press ctrl-C to interrupt execution and once the ebug prompt `>` returns, type `g` to go.

Switch to the Amiga box and type `ft` at the shell prompt to flush all tools from memory. Now quickly switch back to ebug to see tools unloading. When this operation has completed, switch back to the Amiga box and type `asm demo/example/firsthello` at the shell prompt. Quickly switch back to ebug, and you'll see the assembler's tools being loaded.

Ebug Technical Architecture

This section explains a bit about how Ebug works. Ebug is a hosted debugger, so it relies on services provided by an underlying operating system. These services include:

- Creating a new process
- Waiting for a debug event
- Looking at the debuggee's memory
- Looking at the debuggee's registers
- Starting the debuggee running

Linux interfaces can be used to start a new process. When doing this, it's possible to set a flag to indicate that you want to debug the new process you are starting. The debugger process then waits for a debug event of interest to happen in the debuggee process. When a debug event occurs the debuggee process is stopped. The appropriate interface call is then used to look at and change the debuggee's memory and registers while it is stopped. When this has been completed another call is made to restart the debuggee process.

This magic uses the Linux functions `fork` to create a process, `wait` or `waitpid` to wait for a debug event, `ptrace` to look at the debuggee's memory and registers and start it running again.

In theory **ebug** is a general purpose machine-code-level debugger which could debug any Linux executable. However, it has special features which help when the executable is Amiga; it understands the tool structure, Amiga debug information, and Amiga threads.

In theory ebug can run on other flavours of Unix, although currently the Amiga system has only been ported to the Linux variant.

Other Utilities

Other useful utilities are available for debugging and analysis of code. These include the Data Flow Analyser (DFA) and the Test Coverage Analyser (TCA).

DFA

DFA is a data flow analysis utility which can check conformance with the VP2 specification, analyse register data flow and detect anomalies, and emit modified tools containing selected optimisations. DFA currently supports VP tools in small tool format conforming to the VP 2.1 specification.

Using DFA

DFA is designed to be run from the shell. It will accept either a single toolname or filename as input or will accept a series of arguments from **stdin**. DFA output is directed to **stdout**. Consequently its input and output can be redirected arbitrarily.

The command line parameters and usage are summarised here. DFA can be invoked directly or via CAT and a pipe:

```
dfa [-options] toolname[.00]

cat <filelist> | dfa [-options]
```

These are the options:

- | | |
|-----------------|--|
| -b (backup) | backup modified files |
| -d (dependency) | check qcall arguments correspond to referenced tools |
| -f (fatal) | report fatal anomalies only |
| -m (major) | report only major (significant or fatal) anomalies |
| -p (prediction) | insert branch predictions using default heuristics |
| -q (quiet) | suppress anomaly reports |
| -r (redundancy) | identify redundant tags, jumps and unreachable code |
| -s (statistics) | provide summary statistics |
| -t (tabbed) | produce tab-separated summary |
| -v (verbose) | provide detailed anomaly descriptions |
| -w (write) | write optimised tool (zaps added plus other optimisations) |
| -z (zap) | identify opportunities to zap registers |

DFA test categories

Data flow analysis tests and VP conformance checks are performed whenever DFA runs. Other tests or optimisations are only performed when the relevant options are selected. These include the -r, -d, -b and -z options listed earlier.

Additionally, the options selected determine the level of detail and the format of the output; the -f, -m, -q, -t, -v and -s options fall into this category.

If you want to write or 'emit' a modified tool, the -w option must be selected; the modifications made to the tool then reflect the other options selected. If you're not sure about the wisdom of this, -b makes a backup of a file before modifying it.

For example, if you want to find out where zaps can be added to a tool, use the -z option; if you want the zaps to be added automatically, use the -zw option.

As there are a large number of reports generated by DFA it's wise to start analysis using the -fs or -ms combination. If processing a large number of files use the -qt option. This allows the most significant problems to be pinpointed quickly.

Effort spent chasing minor anomalies is usually wasted if there are also significant or fatal anomalies, as correcting the latter will nearly always change the former.

Detailed information about DFA command line options and its operation can be found in `app/dfa/dfa.html`.

TCA

TCA is a test coverage analysis utility that measures the structural coverage achieved when testing VP tools.

TCA currently supports VP tools in small tool format conforming to the VP 2.1 specification.

Using TCA

Measurement of test coverage takes place in three stages:

- instrumenting the object of the test
- running the test
- recording and analysis of the test coverage results

The command line parameters and usage are summarised here. Like DFA, TCA can be invoked directly or via CAT and a pipe:

```
tca [-options] toolname[.00]
```

```
cat <filelist> | tca [-options]
```

- b (backup) backup modified files
- e (entry) instrument tool/method entry points only
- i (instrument) instrument VP binary for test coverage measurement
- w (write) write test coverage data to file
- z (zero) clear historical coverage data

Usage:

```
tca -option
```

Options:

- l (list) list files currently under test
- q (quit) quit tca
- qn (q/nowrite) quit tca without writing coverage data to file
- wa (write all) same as -q

For detailed information about using TCA please refer to `app/dfa/tca.html`.

Instrumenting Object Tools

Instrumenting a tool means modifying the tool binary; the modified tool is functionally identical to the input tool, but has code inserted, which at each entry point and block entry, records the occurrence of execution of that code. When the modified tool is used in place of the original tool, the test coverage data is gathered.

TCA instruments tools using the -i switch. It will accept either a single toolname or filename as input or will accept a series of arguments from stdin. There is no output at this stage unless there are error messages; if a tool has already been instrumented, this will be reported to stdout.

The instrumentation can either be standard, using the -i switch, or for instrument entry points only, with the -ie switch.

Specifying the `-b` switch when instrumenting a tool causes the original to be backed up.

Instrumenting a tool also creates and initialises a test coverage data (`.tcd`) file. This file is maintained for the life of that version of the tool.

Instrumented tools are not intended for inclusion in any deliverable build; they are only intended for the purposes of test coverage measurement.

Running The Test

The test is run in the normal way. An instrumented version of a tool should behave identically to the original; if it does not, this normally indicates that the tool is performing some illegal operations within VP. As the tests are run, a number of named data areas are created and maintained in the Amiga system:

For each instrumented tool, a named data area is created which contains the test coverage data for that tool. The first time the instrumented tool is called, the named data area is created and initialised. Each subsequent call to the tool, and each time any code within the tool is exercised, the relevant parts of the named data area are updated.

There is a single master named data area with a well known name that contains a list header. All of the extant tool named data areas are maintained on a standard Amiga doubly linked list. As instrumented tools are called and their named data areas created, the named data areas are added to the list. Whilst there are any tool named data areas in existence, the master named data area also exists.

Note that the data maintained in named data areas whilst tests are being run is only transient, and only applies to that series of tests. The data must be committed to file to become permanent.

Whilst running the tests, typing `tca -l` in the shell will return a list to stdout of all tools for which named data areas currently exist, if any. This provides a means of discovering which tools are being exercised at any instant.

If a tool is updated and re-instrumented whilst tests are going on, this event will be detected the next time the tool is called, and the named data area re-initialised. There is therefore no need to quit TCA before substituting an updated version of a tool.

Hints and Tips for Programmers

Using the Developer Documentation

A huge quantity of information is available to the developer as part of the standard Amiga build. The documentation consists of a large number of HTML files. Generally, these files have the name `api.html` or `user.html`.

Each file is located in the appropriate directory for the part of the system it is describing. A convenient way of accessing this information is to create desktop shortcuts to `contents.html` and `index.html`. You can then access the documentation using your favourite web browser.

The documentation has an automatic indexing system designed by Amiga. This allows the programmer to use a program called 'help'. Help scans through indexes of the HTML documents looking for certain tagged keywords. Help is a native Amiga program.

Help will give you a selection of pages that have this keyword tagged and you can select the most suitable. It will then display the relevant section of the requested HTML file. This facility is useful for programmers who know the name of a particular tool and wish to check the exact inputs and outputs required.

A simple HTML viewer is available, allowing you to view any HTML file from the Amiga shell. To use the viewer, type `html <filename.html> .`

Tracking Which Tools are Loaded

It is often useful to be able to trace which tools are being loaded by the Amiga tool loader when debugging. This is easily achieved by adding a command line option to the tool loader device driver. To do this, find where the tool loader is started and add the `-l` command line option.

Using the Amiga debug device driver

The Amiga debug device driver allows tracing of method calls to a specified driver. The debug driver can be inserted into the driver stack either between an application and a driver or between two drivers. More information on this useful tool can be found in `dev/dbg/elate/user.html`, in the standard Amiga build.

Translation Procedure

Translation is normally carried out by the system as required. The programmer does not normally need to use the `translate` command directly, but sometimes it can be interesting or useful to do so. This short section discusses the way to invoke a translator manually.

Translators are used in three situations:

- When called by the **sysgen** utility to build a completely new system.
- When called by the tool loader for loading tools (dynamic binding).
- For pre-translation of code using the `translate` command.

The translation of VP code into native form can take significantly less time than that taken to load that code from storage.

The `translate` command can be used to pre-translate a Java class (`.class`) into the corresponding VP tool `.00` files, or to pre-translate a VP tool with a `.00` suffix to a native tool with some other numerical extension, for example `.16` for a big-endian PPC system. Doing this means that system startup and the tool loader use the pre-translated versions, rather than automatically calling the translator.

```
translate [options] <filename> ...
```

For each specified file, a translator is run on the tool contained within that file. The translated tool is then stored in a file with the same name except that an appropriate `.nn` suffix will be added, and any `.00` suffix will be removed. Directories are created as necessary to do this.

The main option to be used with this command is:

```
-t<translator>
```

This instructs Amiga to use the specified translator. By default the current system translator is used. The prefix `sys/tr/` is added.

A more detailed list of translator options appears in *The Amiga Shell Commands Reference Manual*, which is supplied as a PDF file on your development CD.

Architecture and Subsystems

This section provides programmers with an overview of the Amiga architecture, as well as fundamental information about its subsystems. It introduces key concepts. Programmers will need to build on this and improve their knowledge of Amiga by consulting the reference material on CD for more in-depth information.

This section assumes that the reader has already read and understood the previous sections on developing Objects and Developing Tools and is familiar with the basics of assembling and running VP tools. Topics covered in this chapter and those adjoining it include:

- Booting Amiga
- Platform Isolation Interface (PII)
- CPU Isolation Interface (CII)
- Device drivers
- Translators
- Shell

In addition, there is a short section on programming tools, which covers topics such as the Data Flow Analyser (DFA) and debugging with **ebug**.

System overview

This section provides an overview of the Amiga system architecture.

Amiga Modules

The Amiga system comprises three main parts:

- Core subsystems
- Libraries and toolkits
- Development tools

Core Subsystems

There are currently six core subsystems within the Amiga system. More will be added as the system develops.

This is the current set of subsystems:

- Platform Isolation Interface (PII)
- CPU Isolation Interface (CII)
- Device drivers
- Kernel
- Translators
- Shell

Libraries and Toolkits

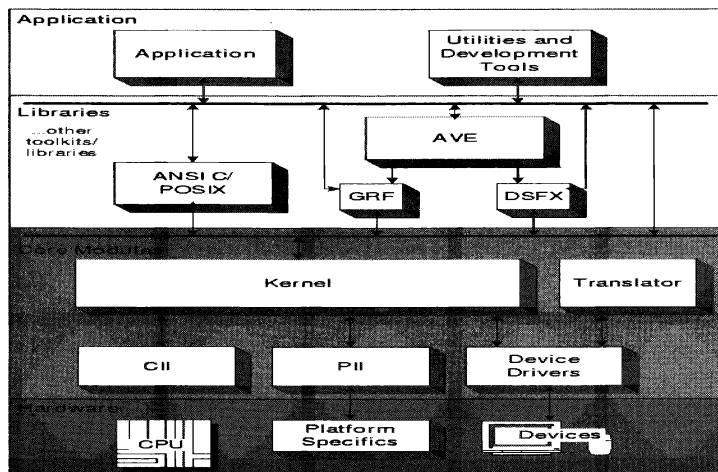
The Amiga libraries consist of sets of ANSI C and Posix compliant functions.

Development tools

- DFA data flow analyser
- Ebug Linux-hosted debugger
- Assemblers for VP and native code
- Editors JOVE, an Emacs clone, and ED, a batch line editor
- Compilers for Java, C and C++

Block Diagram

The following block diagram shows the way the system components fit together.



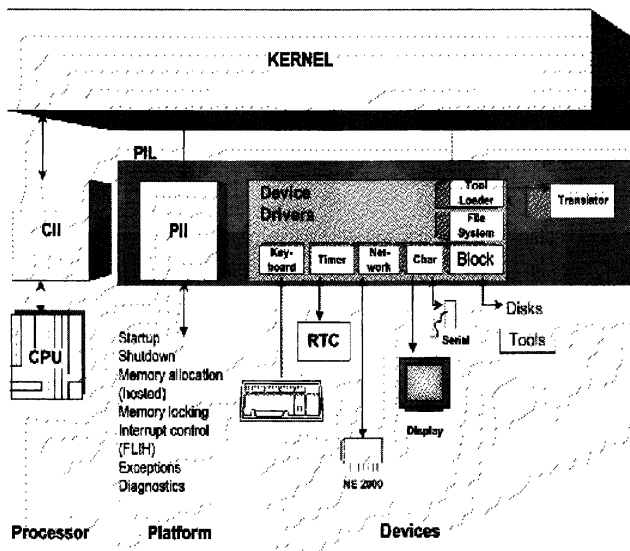
Platform Isolation Interface

The Platform Isolation Interface, or PII, is the layer that allows the Amiga to run on any hardware.

PII Services

The PII provides services for the Kernel and Device Drivers. Some of these are used by one or other, and some are used by both.

The majority of PII tools are written in VP assembly language. PIIs can be divided into two broad types, **hosted** and **non-hosted**. 'Hosted' are those where Amiga runs on top of another operating system. In this case the PII tools very often call the underlying OS to carry out a particular function, such as to lock memory.



The diagram illustrates the logical relationship between the Platform Isolation Interface, the Kernel, the hardware and some of the other Amiga subsystems.

Kernel services provided by the PII

- System start-up tools
- Tools for memory allocation
- Tools for locking/unlocking memory
- Tools for setting exception handlers

Device driver services provided by the PII

- Setting and unsetting of interrupt handlers
- Mapping and unmapping physical address ranges into process address spaces
- Getting physical addresses of logical addresses

Kernel and device driver services provided by the PII

- Returning the address of the host parameter string
- Enabling and disabling interrupts

PII Tools

What follows is a list of the most common PII tools, with a brief description of their function. For more information refer to the *PII Reference Manual*, within the Amiga build at `sys/pii/`. If you are tempted to write a PII please also refer to the hosted PII Construction Guide in the `sys/pii/` directory.

System Startup and Shutdown

`sys/pii/boot` - the boot tool is responsible for booting Amiga into a runnable state
`sys/pii/shutdown` - shuts down the Amiga system, releasing all allocated resources

Information Provision

`sys/pii/info` - returns a PII type and a pointer to a PII specific block of memory
`sys/pii/stksize` - returns the minimum extra stack space required for the PII to run

Memory Allocation

`sys/pii/alloc` - calls host OS to allocate memory
`sys/pii/free` - calls host OS to free memory

These only apply to the hosted configuration; if there is no host Memory Allocation is handled entirely by the Amiga system.

Memory Management

`sys/pii/lock` - locks a memory region to ensure it will not be paged out to disk
`sys/pii/unlock` - unlocks a region, which becomes eligible for paging out to disk
`sys/pii/map` - maps a physical space to the address space of an Amiga process

sys/pii/unmap - unmaps process address space so that memory resources are freed
sys/pii/get_phys - gets the physical address of a memory region
sys/pii/makeexec - makes a memory region executable
sys/kn/mem/gwc - deterministic best-fit allocation and constant-time freeing

Interrupt Management

sys/pii/setint - allocates an Interrupt Service Routine (Second Level Interrupt Handler) to an Interrupt Request - supports multiple routines on one Interrupt Request, like the Classic Amiga chained interrupts
sys/pii/unsetint - removes Interrupt Service Routine from an Interrupt Request
sys/pii/int_dis - disables the specified Interrupt Request
sys/pii/int_en - enables the specified Interrupt Request
sys/pii/int_off - disables all hardware interrupts
sys/pii/int_on - enables all hardware interrupts, except those specifically disabled
sys/pii/int_restore - restores the interrupt state to that returned by sys/pii/int_off
sys/pii/sched_op - cause scheduler operation to be carried out as soon as possible (as the First level interrupt Handler returns to the scheduler)
sys/pii/halt - halts Amiga allowing host OS to take control for power management

The above routine is called by the kernel after interrupts are switched off - the halt should at least switch interrupts on and return.

Exception Management

sys/pii/setexc - allocate exception handler routine to handle specified exception
sys/pii/unsetexc - deallocate exception handler from specified exception

The **unsetexc** call uses the handle previously returned by sys/pii/setexc.

Host system specific

sys/pii/opentool - opens and binds host-specific code to an Amiga tool
sys/pii/closetool - deallocate resources associated with host tool binding
sys/pii/hostparms - return address of host parameter string
sys/pii/stkvars - setup platform-specific stack on hosted PII's
sys/pii/threadevent - notification of hosted thread creation or destruction

Process Synchronisation

sys/pii/swap - swap a non-zero integer and return result

Diagnostic Functions

sys/pii/odata - output diagnostic string to host system (platform specific)
sys/pii/ktrace - output diagnostic string with numerical data (calls odata, portable)
sys/pii/trdata - hex dump of memory to trace system (calls odata, portable)

Device drivers

Introduction to device drivers

Amiga device drivers comprise classes written in VPcode that control hardware devices directly. To carry out certain functions the drivers call upon the services provided by the PII. For example loading interrupt service routines is indirectly handled by the PII.

The device driver hierarchy is strictly class based, with all device drivers inheriting from `dev/class`. This means that drivers for a device family can quickly be created by subclassing, the subclasses having hardware specific details where required.

This class-based approach has additional benefits; Amiga device drivers have a very well defined set of methods that need to be implemented, this in turn means that the device driver software interface is consistent and easy to use. It is also relatively easy to extend existing drivers or develop new drivers.

Because there is no supervisor or user mode in Amiga, device drivers can be called directly from user applications. It is therefore possible to start, stop, load and unload drivers at run-time, just as you could with a Classic Amiga. Note that unlike some nasty monolithic operating systems (no names, no pack drill!) the Amiga device drivers are not compiled into the kernel.

Amiga device drivers are usually interrupt driven. As will be explained in more detail later, an interrupt service routine is installed as part of the driver's init method.

Alternatively, input and output can be achieved with Amiga libraries, such as the ANSI C library. These libraries in turn call device drivers for low level input and output.

Writing Device Drivers

Where programmers need to write new device drivers to support special or new hardware, they are strongly urged to read *The Device Driver Design Guide* which we supply as a CD file, and consult developer support (support@amiga.com) to make sure that they're not trying to reinvent a wheel. The PDF document has extensive details on how to write drivers. This manual considers the issues that arise in developing drivers which are relevant to most programmers.

Memory Mapping

The PII provides native tools that are responsible for mapping areas of physical memory into an Amiga process's address space. Device driver programmers will require the ability to map between physical and logical memory.

Access to Input/Output (I/O) Ports

Access to hardware I/O ports should be achieved through use of the `sys/cii/ioin` and `sys/cii/ioout` macros.

Exclusive Software Resource Access

Typically, data and information about a specific device is held in the device object's instance data. This data should only be accessed by one process at any one time, otherwise serious corruption of the devices' data integrity could result. In order to protect the instance data in a driver object, the data is protected by a **mutex** - MUTual EXclusion - scheme. The instance data can be protected by calling `sys/kn/mtx/lock` before accessing the instance data in driver method code such as `read` and `write`.

Page Out Prevention

It is important that routines such as interrupt handlers are not paged out to disk. This could potentially occur when Amiga runs on a host OS that performs virtual memory management by paging. In order to prevent this you should use the tools `sys/kn/mem/lock` and `sys/kn/mem/unlock`. These routines will call the necessary native tools `sys/pii/lock` and `sys/pii/unlock` in the PII.

Exclusive Hardware Resource Access

On initialisation it may be advisable to lock the I/O registers of a device. This can be achieved by using the tool `dev/lockio`. This tool can also be used to book an Interrupt Request ID.

Device Driver Read/Write Policy

It is also important to decide on the device driver policy that will be implemented. Currently there are two possible policies: synchronous or **blocking** and non-blocking access, also known as **asynchronous**.

The Amiga team develops device drivers to implement both policies. The differences between these policy types are discussed in more detail below.

Device Driver Methods

The method API for a device driver is carefully defined. Below are listed some typical methods. Drivers may not support all these methods; for example, a keyboard driver would not have an implementation for the write method, for obvious reasons.

These methods are implemented by all device types:

- _alias*
- _deinit*
- _init*
- close*
- defaultmethod*
- info*
- open*
- read*
- reada*
- reference*

These are additional methods implemented for block devices:

- bsize*
- clearerror*
- flush*
- seek*
- sync*

These are additional methods implemented for character devices:

flush
getflags
setflags
status
sync

These are sufficiently general that there are no additional methods implemented for pointer or keyboard devices.

Other methods are easily added as required for specific devices, so a CD-ROM driver might have these additional methods:

cdeject
cdload
cdplaytrack

Device Driver allocator and de-allocator tools

Where classes are going to be instantiated they normally have two special tools called `_new` and `_delete`. These tools allocate and de-allocate memory for the objects and providing an object pointer which can be used to invoke the device driver's object method code. Device drivers have slightly different code in their `_new` and `_delete` tools when compared to 'non-driver' classes. This is because device drivers must allow for a doubly-linked list node before the instance data. This is used by the system to keep track of the driver instances.

Here's some example code:

```
tool 'dev/example/_new',VP,0
ent - : p0                ; return instance pointer
cpy.i MH_SIZE+DEV_SZ,i0
qcall sys/kn/mem/allocdef,(i0:p0,i~)
if.p p0 != NULL
    add.p MH_SIZE,p0
    refclass p0,dev/example/class
endif
ret
toolend
```



```

tool 'dev/example/_delete',VP,0
ent p0 : -
derefclass p0
sub.p MH_SIZE,p0
qcall sys/kn/mem/free,(p0 : -)
ret
toolend

```

Workings of an `_init` Method

Normally a driver `_init` method would perform the following tasks:

- initialise instance variables and override these with values from command line parameters if present
- initialise a mutex in preparation for locking instance data in read and write methods, using `sys/kn/mtx/init`
- lock the I/O address range for the device using `dev/lockio`
- book an Interrupt Request line for use by the device, also using `dev/lockio`
- load the device Interrupt Service Routine with `dev/loadisr`
- set-up the hardware device by disabling interrupts with `sys/pii/int_off`, writing to the hardware or check status using `dev/out` and `dev/in` tools, then switching interrupts back on using `sys/pii/int_restore`

The `_init` method will often need to create a parentclass object before the subclass can be instantiated. If for some reason the parentclass cannot be created then this must be handled in the subclass `init` method.

If the parentclass is created successfully, but for some reason the subclass object fails to initialise, this case should be trapped and the parentclass object deinitialised. There is a fairly standard format for handling these situations across different device drivers, so code can readily be shared.

Workings of a `_deinit` Method

The `_deinit` method typically reverses the procedure carried out by the `_init` method. This involves deallocating any buffers that may have been allocated, deinitialising mutexes, disabling interrupts, unlocking memory areas and unloading Interrupt Service Routines. In addition the subclass object will need to ensure that the parentclass object is deinitialised.

Workings of a read Method

In essence, a read method would:

- lock instance data with `sys/kn/mtx/lock`
- read data from the device
- unlock instance data with `sys/kn/mtx/unlock`

Workings of a write Method

A simple write method would:

- lock instance data with `sys/kn/mtx/lock`
- write data to the device
- unlock instance data with `sys/kn/mtx/unlock`

The Interrupt Service Routine

If the device driver is interrupt driven, when the device driver is initialised an Interrupt Service Routine is loaded. The actual operations carried out by this routine are device-dependent.

This routine is also known as a Second Level Interrupt Handler. You probably will not be surprised to hear that it is called by the First Level Interrupt Handler...

The First level Interrupt Handler is normally coded in native assembler and is part of the PII; for the PC stand-alone platform the First level Interrupt Handler is found in the tool `sys/pii/pcstand/flih`.

The tools `sys/pii/setint` and `sys/pii/unsetint` are actually responsible for attaching and detaching Second Level Interrupt Handlers to the First level Interrupt Handler, although `dev/loadisr` provides a convenient alternative for device driver work.

Multiple Second Level Interrupt Handlers on one Interrupt Request are supported for the situation where several devices share an interrupt ID. The PII chains these handlers, so that once an Interrupt Request has occurred each Second Level Interrupt Handler for that interrupt ID is called in turn.

The Second Level Interrupt Handler is coded so that it can determine whether it is actually meant to handle an interrupt from that device or not. If it does handle the interrupt it may need to wake up any process waiting on this event.

Blocking, Non-blocking and asynchronous policies

There are generally three types of I/O policy in the Amiga, termed blocking, non-blocking and asynchronous (AIO). The choice amongst these has implications for the efficiency of the entire system. In all three I/O policies the driver code is ncalled from the application, and thus runs in the context of the application's process.

This scheme resembles 'Quick I/O' on Classic Amigas, and unlike the normal situation when I/O is performed by other processes on behalf of the caller. It follows that you may need to allocate extra threads for I/O to gain equivalent performance when porting Classic Amiga applications to the new system.

Non-blocking

If a driver is opened in this mode and an I/O operation such as read is performed, the read will return immediately, indicating that the operation has completed with some data, if any is available. If the operation could not be completed at this time, a value indicating this is returned to the calling code.

Blocking

Here the device driver blocks until the I/O operation completes. To achieve greater efficiency the driver will be expected to sleep after initiating the I/O operation. The driver can then be woken by the Interrupt Service Routine to complete the I/O using a call to `sys/kn/int/proc/wake`.

Asynchronous I/O

This is the most complicated form of I/O; there are several parts to this system:

- calling application
- notification mechanism
 - callbacks
 - event flags
 - signals
- AIO structure
- device driver
- interrupt service routine

Notification mechanism

Notification provides a mechanism whereby the calling application can be made aware that an I/O operation has now completed and that the application code to process this can now be run. The following section discusses callbacks as this notification mechanism is commonly used in the Amiga system.

A callback handler is a subroutine that can be written at the application level to process the result of an I/O operation. The callback routine does whatever the application needs to do once the I/O operation has been completed by the underlying layers. For example, in the Audio Visual Environment, if data becomes available from the keyboard, the callback will dispatch a keyboard event to the target Audio Visual Object.

AIO structure

The AIO structure is used to store important information about the AIO operation. Each AIO operation requires an AIO block which cannot be reused until after the AIO has completed.

AIO blocks make it possible for the driver to manage a number of simultaneous outstanding I/O operations. The structure of the AIO block is described in the device driver documentation on your development CD.

Interrupt service routine

The Interrupt Service Routine performs initial processing on the incoming interrupt. Before returning, the Interrupt Service Routine will call `dev/iocomplete` to complete the I/O operation, using a pointer to the AIO block of the calling application's process as a parameter. This flags the callback routine registered by `dev/ioprepcallback` to be processed by the kernel.

Sequence of operations for asynchronous I/O

The application calls a `reada` or `writea` method on the device object. The driver code returns to the application immediately having initiated an I/O operation. The application code is expected to sleep at this point, effectively waiting for completion of the I/O without using processor resources. Once the I/O operation is completed by the hardware an interrupt occurs, which is processed by the installed Interrupt Service Routine. The Interrupt Service Routine calls `dev/iocomplete` when the initial interrupt processing is done.

dev/iocomplete flags a callback to be processed by the kernel by calling sys/kn/callback/setflag. The callback is registered to run at the application process level, although it is possible to register the callback to run at the device driver process level, if there is one. This is achieved using dev/ioprepdevcallback.

Device driver method implementation

The device driver developer only needs to code asynchronous method calls when implementing a new driver. This is because the driver base class dev/class fully implements methods such as read, write, status etc. by using calls to reada, writea, statusa, which are implemented further down the class hierarchy. The base class adds wrapper code around these calls to the asynchronous methods to implement the blocking of the read, write and status methods, where required.

This leaves the device driver developer free to concentrate on developing the asynchronous methods without having to worry about handling synchronous cases.

Example code

Application

```
dev/ioprepcallback, (aio_ptr, callback_handler, \
    callbackdata_ptr)
ncall dev_ptr, reada, (dev_ptr, dev_handle, \
    callbackdata_ptr, aio_ptr, bytes_to_read: flags)

loop
    qcall sys/kn/proc/sleep, (-1.1:I~)
endloop
```

The backslashes indicate lines which have been wrapped to fit the printed page.

Callback handler

```
callback_handler:
    ; p0 = aio_ptr
    ; p1 = callbackdata_handler

    ent p0 p1:-

    qcall dev/iogetresp, (aio_ptr: status)
    ; get final status of operation
```

```

qcall dev/ioprepcallback, (aio_ptr, \
    callback_handler, callbackdata_ptr)
ncall dev_ptr, reada, (dev_ptr, dev_handle, \
    callbackdata_ptr, aio_ptr, bytes_to_read:flags)
ret

```

Interrupt service routine

```

<handle the incoming event>

dev/iocomplete, (aio_ptr, status:return_status)
ret

```

Extensive interrupt processing

If the Interrupt Service Routine needs to do more extensive processing of the incoming interrupt then an alternative approach is necessary. This is because in Amiga interrupts are disabled while in Interrupt Service Routines. This could potentially result in interrupts not being processed in a timely manner. The Amiga solution for handling nested interrupts is to run the code to handle the interrupt in a separate, high priority process. This process needs to be of a high priority to ensure timely execution by the kernel and ensure that interrupts are not missed.

The code to implement the high priority process is actually inherited from the device driver base class `dev/class.asm`. The process is coded as a main tool, which the driver can spawn automatically if required. This provides a high priority context in which a callback handler registered at the device driver level can be executed. This driver level callback handler could perform a `dev/iocomplete`, flagging up a callback handler registered at the application level to execute, within the application's context.

Starting and Stopping Device Drivers

There are three ways to start device drivers:

- using `.obj` in a `sysgen .sys` file
- using the shell command `devstart`
- under program control

All three techniques essentially do the same things:

- create a device driver object
- initialise the device driver object
- add the device driver object to the mount table

At any stage the device driver mount table can be displayed by the shell command:

```
$ devinfo
```

If information on a specific driver is required, devinfo can be used in conjunction with the driver name:

```
$ devinfo /device/mouse
```

Note that the \$ character is merely the shell prompt and is not part of the command.

Using devstart

To start a device driver from the command line you will need to use devstart followed by the required parameters. The basic syntax is `devstart <device name> <device driver> [other parameters]`, for example.:

```
$ devstart /device/mouse dev/mouse/pcserial
```

To unload this device driver you would type:

```
$ devstop /device/mouse
```

You must specify the driver name, as there may be several devices sharing the same driver object. If additional parameters are not provided then the driver will use its built-in defaults.

Loading Device Drivers From Software

This can be done inside programs by using the following sequence of instructions:

- call the device driver allocator tool `_new`
- invoke the driver object `_init` method
- call `sys/kn/dev/mount` to add the device to the mount table

Accessing Drivers From an Application

Assuming the device driver has already been successfully loaded with the `devstart` command, perhaps from within a script, or via an `.obj` command from a sys file, this is the procedure for accessing the driver from your application:

- Use `sys/kn/dev/lookup`
- Open the device
- Invoke methods on device as required by the application
- Close the device

Each of these steps is explained in more detail below:

`sys/kn/dev/lookup` takes a string as its input. This string is the logical name of the device. Examples of logical names include:

```
/device/loader  
/device/error  
/device/null  
/device/rwfs
```

These logical names appear in the mount table, which can be checked any time with the `devinfo` shell command. `sys/kn/dev/lookup` will scan through the mount table looking for the best match to the string you provide.

`sys/kn/dev/lookup` will then return with two pointers:

- A pointer to the device driver instance, or `errno` if it fails
- A pointer to a residual string

The instance pointer is subsequently used to `ncall` device methods as required in the application. The residual string is the difference between the string provided by the application and the closest matching logical name found in the mount table.

For example, assume the device driver defined by `dev/example/class.asm` is loaded and mounted as `/device/example`. If the programmer then performs `sys/kn/dev/lookup` providing `dev/example/balti` as the input string then the residual string would be `balti`. For an exact match the residual string will be blank; the programmer can check for this if an exact match is required.

The purpose of the residual string is to provide access to a resource within the driver. What this means in practice depends on the actual device and is discussed in the user documentation for the associated driver. As an example, the Sejin

keyboard driver uses the residual string as a way of selecting the mode of operation of the keyboard. In the file system the residual string is used as the file name for the file to be used as storage

In practice `sys/kn/dev/lookup` is unlikely to fail, as it is very difficult for it not to find a partial match in most cases. The residual string should therefore always be checked by the programmer. There might be rather more 'residue' than you expect.

The open method

This method is invoked to open the device and return a handle to it. This handle is typically used in subsequent `ncalls` to the driver methods. The returned handle should be checked by the programmer to ensure the device opened successfully. The `boolerrno` and `iferrno` macros are particularly useful for this purpose.

The device may fail to open because another application is already using it. Some device drivers can actually have one instance with multiple handles. A good example is the floppy disk controller, which can have one instance with four handles - one handle for each of the drives it can control.

Once the device is open, you invoke methods as required - for example:

```
ncall p0,info,(p0,p1,p2,i0:i0)
```

The close method

This method closes the device, but does not delete the device from memory or from the mount table. Here is an example call:

```
ncall p0,close,(p0,p1:i0)
```

Device Families

The Amiga device architecture currently supports these device driver families:

```
structure
byte DF_RESERVED ;Not used
byte DF_UNDEFINED;Unknown or undefined devices
byte DF_BLOCK      ;Block devices (hard drive etc.)
byte DF_CHAR       ;Character devices (e.g. Serial)
byte DF_SOUND      ;Sound devices (e.g. WAV)
byte DF_KEYBOARD   ;Raw keyboard devices (e.g. Keypad)
byte DF_POINTER    ;Pointer devices (e.g. Mouse)
byte DF_FILESYS    ;File System devices (e.g. FAT)
byte DF_TABLET     ;Tablet devices (e.g. penpad)
byte DF_PROTOCOL   ;Protocol devices (e.g. TCP/IP)
byte DF_NETWORK    ;Network devices (e.g. Ethernet Adapter)
byte DF_MESSENGER ;Messenger driver
byte DF_LINK       ;Link devices
byte DF_GRAPHIC    ;Graphic devices
byte DF_AVE        ;Audio Visual Environment devices
byte DF_3DGRF      ;3-D GRF services
byte DF_TIMER      ;Real Time Clock timer devices
byte DF_JOYSTICK   ;Joystick device
byte DF_CPLOADER   ;Coprocessor loader
byte DF_CPDISPATCHER ;Coprocessor dispatcher
```

The `info` method returns the device family as part of its information packet.

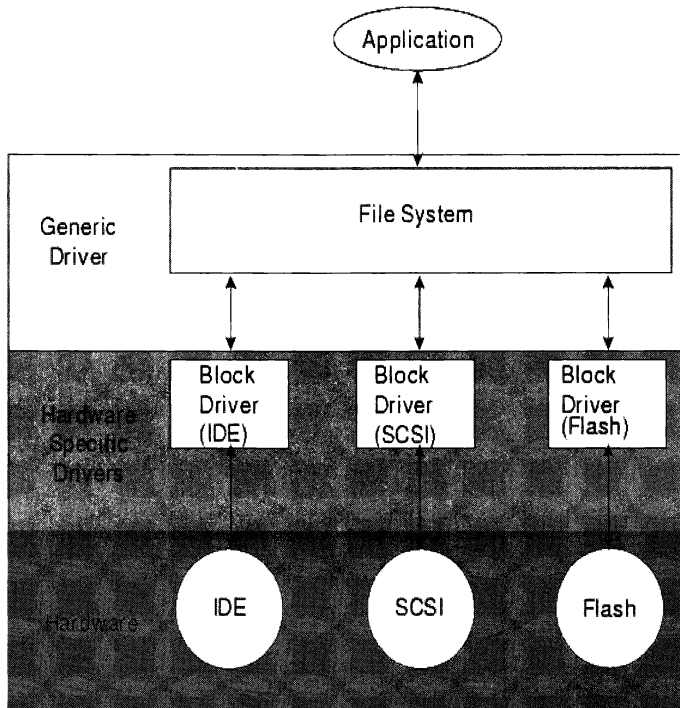
The command `devinfo` uses the device driver's `info` method to return this information:

```
19990823:/$ devinfo /device/rwfs
/device/rwfs:
  Family: file system
  Caching Longname Driver Version 1.24 over Win32 File
    System Driver Version 1.22
19990823:/$
```

Available Device Drivers

Device Driver Types

Device drivers can be classified as either hardware-dependent, if they control hardware directly - this makes them device and platform specific - or generic, in which case they are portable across platforms. The relationship between generic and hardware-dependent drivers is illustrated in the following diagram.



Classification of drivers

Drivers can be further classified according to the type of device they control, the main categories being:

block	keyboard	pointer
character	link	protocol
file system	messenger	sound
graphic	network	timer

- For example, consider the device drivers available for a stand-alone PC clone (ugh) with no host operating system. The currently available drivers include:

Hardware device	Driver type	Description
Real-time clock	Timer	Uses RTC periodic interrupt
PS2 keyboard	Keyboard	Later IBM-style keyboard
Character display	Character	80x25 text mode display driver
IDE controller	Block	Integrated Drive Electronics
SCSI controller AIC 78xx	Block	Adaptec AHA 2940 SCSI
Floppy disk controller	Block	Western Digital, VLSI etc.
PS2 mouse	Pointer	Later IBM-style pointing device
Serial mouse	Pointer	PC afterthought pointing device
Serial com port	Character	16550 buffered UART or compatible
Parallel	Character	Centronics printer or similar PIO
Sound Blaster 16	Sound	Stereo digital to analogue converters
NE2000 controller	Network	Ethernet adapter device driver

There are many more drivers available for other platforms such as QNX, OS/9, Windows, MSDOS, plus various flavours of Unix and device drivers for single board computers, set-top boxes and network computers.

Generally speaking, a hosted version of a device driver will simply make a call to the underlying OS in order to carry out a device-related activity.

Generic Device Drivers

Amiga also includes a number of generic device drivers, which are portable across many different platforms. They normally use the service of an underlying hardware specific driver. Amiga's generic drivers are discussed in more detail in the remainder of this section.

NULL device (character)

This device throws away anything you send to it, rather like the Classic Amiga NIL: device.

Trace device (character)

This device provides system tracing facilities.

Keyboard cooker device (character)

This takes raw key up and key down codes from the keyboard device driver and cooks these to an appropriate character code.

Tool loader

The tool loader uses an underlying file system driver. Its job is to find tools on disk and load them, calling the translator if required. The translator is given a VP tool and returns a native tool.

ZIP file system

This file system enables a compressed or uncompressed Zip file to appear to Amiga as a read-only file system. The Zip file may be in ROM or as a file on an underlying file system.

File block driver (block)

This uses an underlying file system driver. Enables a file to appear as a block driver. It can also provide services to the Amiga file system.

Partition driver (block)

This uses an underlying block driver. It manages partitions on a disk by making each partition appear as a separate block device.

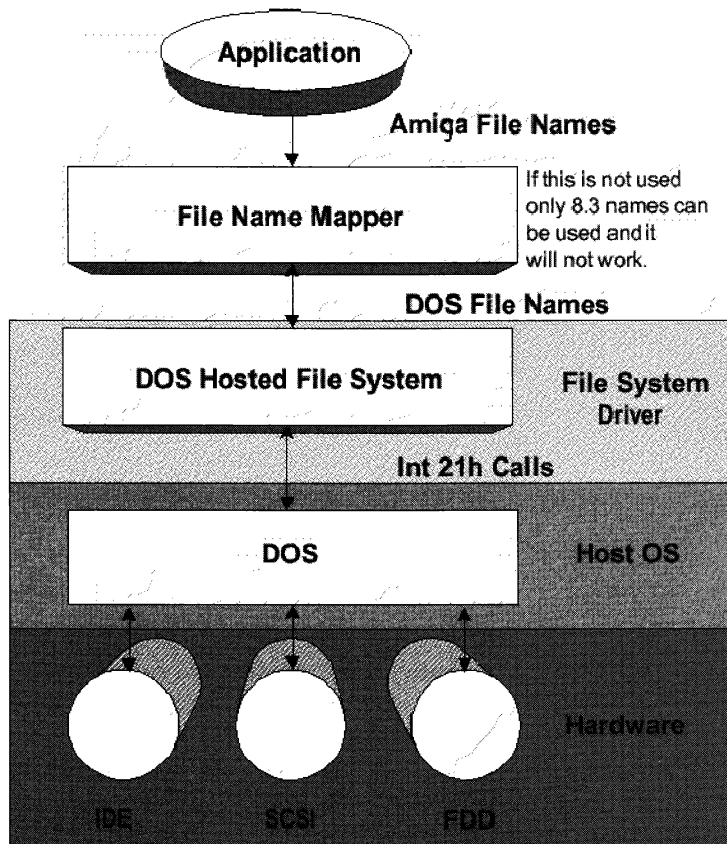
Character link (link)

These links use an underlying character driver. They are employed in in Amiga messaging between processors.

File name mapper (file system)

This uses an underlying file system driver. It is used where Amiga is running hosted by another OS. The mapper provides a mapping between Amiga format file names and those of the underlying file system. The mapper handles issues such as character codes, file name length, case sensitivity and case preservation.

The next diagram shows an application of the file name mapper driver, persuading the Amiga system to run despite the handicap of the 8.3 file naming convention of CP/M and MSDOS (here referred to as DOS):



PPP (network)

Uses an underlying modem driver. Implements the Point to Point Protocol used to obtain dial-up access to a TCP/IP Internet.

SCSI disk (block)

This uses an underlying SCSI control driver.

Modem (network)

This uses an underlying serial driver, and drives a modem by passing it the necessary Hayes AT commands.

FAT file system (file system)

FAT uses an underlying block driver. FAT12 and FAT16 are supported. The FAT file system should be used in conjunction with the file name mapper driver in order to create an Amiga compatible file system.

Messenger (messenger)

This uses an underlying link driver. Messengers communicate between Amiga processes that are running on different physical CPUs. In other respects they work like pipes.

Pipe (character)

This implements both named and un-named pipes which are local to the Amiga, making them faster than messenger connections.

TCP/IP (protocol)

This uses an underlying network driver.

Merge file system (file system)

This enables a read-only file system to be used in conjunction with a read-write file system, providing read-write access to the files. It works by invisibly copying files from a read-only file system to a read-write file system where necessary.

PCI

This manages devices on a PCI (Personal Computer Interconnect) bus.

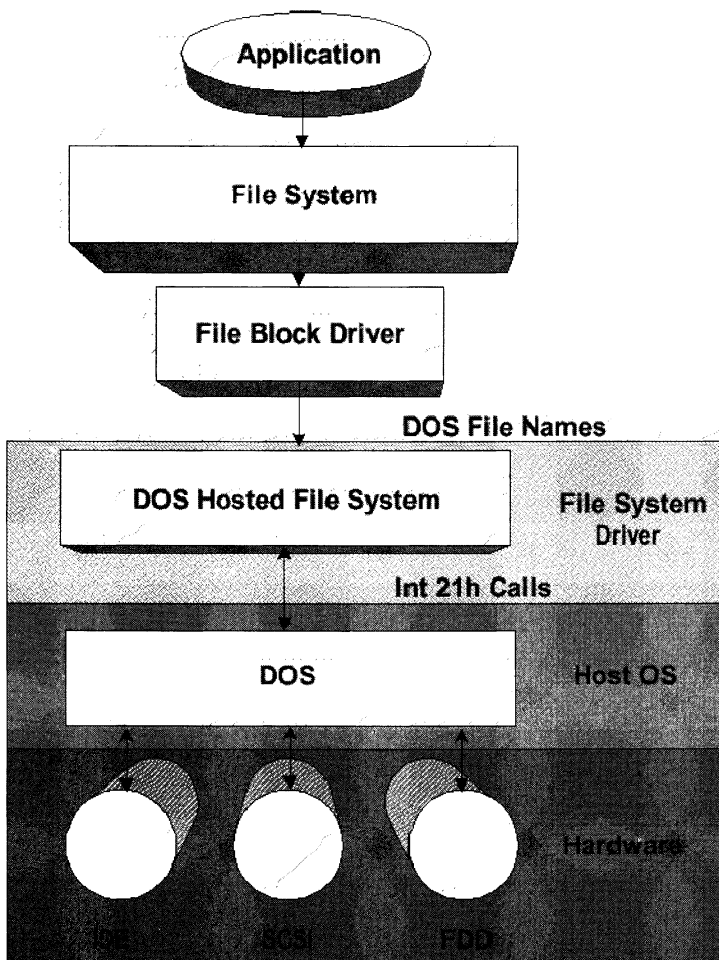
Block cache driver 2.0

The block cache driver is used in conjunction with the FAT file system. On hosted systems Amiga indirectly uses the caching facilities of the host OS.

Amiga filesystem (filesystem)

This uses an underlying block driver. The Amiga file system is designed to be used on Flash ROM devices and on host file systems via the file block driver.

The following diagram illustrates a DOS hosted application of the Amiga file system. Note that the relatively slow file mapper driver is not required in this implementation, because the direct use of blocks bypasses the limitations of the MSDOS file names. While it's unlikely that we'll try to host Amiga on MSDOS, or CPM for that matter, the technique could still be useful for other systems.



Layered keyboard (keyboard)

This uses an underlying character driver. This driver is to support a keyboard connected to a remote system, where the remote system is not running Amiga. The underlying character driver is a serial driver.

Layered mouse

This uses an underlying character driver. This driver is to support a mouse connected to a remote system, where the remote system is not running Amiga. The underlying character driver is a serial driver.

Layered pen

Uses an underlying character driver. This driver is to support a pen device connected to a remote system, where the remote system is not running Amiga. The underlying character driver is a serial driver.

Display Device Drivers

Amiga supports character-oriented displays, for text, and pixel-oriented graphics.

Graphics Displays

Graphics display device drivers implement the interface documented in `dev/grf/api.html`. This is based on the concept of a **pixelmap** and contains methods for creating and destroying screen pixelmaps.

The content foundation includes the Multimedia Toolkit and Graphical User Interface. It integrates event handling, graphics, windows, gadgets and sound, with minimal overhead.

Text Displays

The text display device driver offers character oriented access to a display. The application program interface is documented in `dev/display/api.html`. The driver is accessed through `open`, `write` and `close` methods. The `write` method supplies a sequence of commands to the display device driver. These are documented in `dev/display/sequences.html`.

VT52 escape codes provide backwards compatibility with legacy applications and ports of UNIX applications. The original VT52 was a bulky terminal made by DEC with a built-in electrostatic printer, with the ingenious though messy feature that it only worked if you regularly poured salt solution into the mechanism. This functionality is not supported by Amigas!

Our new custom escape sequences are designed to be more efficient and more flexible than traditional VT52 commands. They also work without added salt.

Character Codes in Amiga

The packet returned by the keyboard device is a single 32 bit integer. The API for raw keyboard devices specifies this integer as an Amiga raw keycode. These raw key codes are defined in the file `dev/keyboard/keyboard.inc`. Bit 31 of this raw keycode is set if the packet indicates the release of the key. The remainder of the keycode is a valid Unicode character.

Keys that cannot be represented by Unicode characters, such as the cursor keys, have codes defined in the user space of Unicode. The Amiga codes are between 0xE000 and 0xEFFF. This integer format is somewhat inefficient in terms of memory usage; for that reason the code is often converted into a multi-byte character format which is relatively memory-efficient.

A keyboard device will usually make no attempt to maintain the shift-state or provide keyboard cooking, so the Unicode character will refer to the unshifted state. A separate keyboard cooker is needed if the shifted states are required. The standard system keyboard cooker documentation has more information, in the file `elate/dev/keyboard/cookdev/user.html`.

The keyboard cooker is a wrapper for the `dev/keyboard/cooked/class`. The keyboard cooker performs additional tasks such as processing only typed events and managing shift states, according to its installed cookery table. The Amiga POSIX shell, for example, uses the keyboard cooker driver to convert integer Unicode into multi-byte encoded Unicode characters.

The Multimedia Toolkit does not use the keyboard cooker driver to convert to multi-byte format. It utilises the `dev/keyboard/cooked/class` directly in order to process shift-state and generate up/down/typed events. These events specify a 32-bit integer Unicode character plus shift-state.

The multi-byte encoding standard used in Amiga is UTF8. This encoding scheme is also supported in the ANSI C library functions. When this scheme is used the least significant byte is equivalent to a seven bit ASCII code.

Module Sizes

One important factor in the design of the Amiga system has been the need for concise component modules, so it can be efficiently incorporated into devices of all sizes. At the moment we're using a big resource-hungry generic computer as a development system, but this is a temporary expedient. We need to be able to squeeze the Amiga into any niche - especially those where lesser systems will not fit. The section demonstrates how we're going to do it.

The VPCode sizes of some typical modules are tabulated here:

Subsystem	Size in Kbytes
Complete ANSI/POSIX library	80
Pentium translator	88
Kernel	62
Platform Isolation Interface (PII)	10
CPU Isolation Interface (CII)	8
Shell	69

These components provide access to Amiga's content foundation. This is a platform combining both multimedia tools, gathered together into the multimedia toolkit, and the engines needed to run the content, such as Amiga Java.

The VPCode sizes for an implementation of Amiga Java measure up as follows:

JBC subsystem	Size in Kbytes
Jcode translator	44
java.applet	6
java.awt plus java.awt.peer	201
java.awt.datatransfer	12
java.awt.event	41
java.awt.image	32
java.beans	53
java.io	146
java.lang	168
java.lang.reflect	48
java.net	63
java.security	26
java.text	75
java.util	85
java.util.zip	44

VP is usually more compact than native codes and so a bloat factor has to be allowed for in order to estimate the actual ROM image size.

For embedded applications where static binding is used, so that all tools are pre-translated and bound into a system image file, the bloat factor is around 1.2.

This assumes that support for dynamic binding has been stripped from the tools. If dynamic loading and binding of tools is required then the effective bloat factor is typically higher, at around 1.5 to 1.7.

Taking these figures into account and allowing 50k for device drivers and miscellaneous tools, a typical set-top box style deployment could be approximately:

Amiga run-time	519K
Java library	1044K
Bloat factor	x1.2
Total	1876K

That comes to less than 2 MB of native memory for the ROM image containing the Amiga system and Java. In addition to the ROM requirements, the system will require RAM. To obtain an estimate of the amount of RAM required, add 368K to the actual code size.

The 368K figure is derived from examining existing systems utilising Amiga. This RAM figure assumes that Amiga is being executed directly from ROM. Note that the Amiga system can execute directly from ROM without needing to be loaded into RAM first.

The above figures are just an example. Amiga systems are most unlikely to use a Pentium, for example, so the size of the translator will be different depending on the choice of processor.

Actual figures will also vary depending on the specific platform solution. The important point to note is that this is a system with a small footprint, yet with all the functionality and extensibility required for modern interoperable systems.

The Amiga Shell

The Amiga shell is a scripting command language interpreter. It is able to read and execute commands from the user, providing an interface to the underlying Operating System.

Commands are entered at the \$ command line prompt, which are then dealt with by the command processor, which calls on the services provided by the Amiga kernel as required. Input and output are redirectable, and the results of most commands are displayed to standard output, which defaults to the screen. The Amiga shell has a similar feel to standard Unix shells, and functionality comparable to a zsh shell, but is far less greedy for resources.

Using The Amiga Shell

All `stdio` programs can be run from the shell prompt. **stdio** stands for 'standard Input and Output'. All the commands described here can be found in the `app/stdio/` directory, although it is quite possible to run external commands that don't exist anywhere in the filesystem. The shell searches all directories in the path `$shell.path`, which is normally set up to contain only `/app/stdio`.

For further information upon the Amiga Shell Commands please see *The Amiga Shell Commands Reference Manual*, which is supplied as a PDF file on your Amiga Development CD.

Commands can be entered at the \$ prompt, displayed on the screen within the Amiga box. Where you see angle brackets <thus> the contents of the example should be replaced by an actual parameter when typing a command. Parts of the command line shown in square brackets [thus] are optional. An ellipsis indicates that more of the same may follow.

So a command with from one to three parameters, or more, would be written:

```
command <parameter1> [<parameter2> [<parameter3> .. ]]
```

In this case parameter 1 is mandatory, and parameters 2 and 3 are optional. However, parameter 3 cannot be specified unless parameter 2 also is. Parameter 3 may be repeated.

If a dollar sign appears at the start of a line, this is a hint that it is a command. The dollar sign is the shell prompt, not actually a part of the command that you type.

Command Options

Commands can be modified by specifying additional options. All options must be preceded by "-". Multiple options can be specified together, so for example, "-abc" would behave identically to "-a -b -c". However some options may take additional arguments, in which case multiple options cannot be specified. Options can appear anywhere on the command line, between parameters. For clarity however, it is recommended that options be placed immediately after the command name, and therefore before any parameters.

Example Commands

The command names follow Unix conventions, which means they seem arcane at first - the names are chosen to be short, easily typed and distinguished, rather than for their obvious mnemonic significance. They seem simple but this is deceptive - they are carefully chosen and surprisingly flexible in practice.

Listing Files with ls

The `ls` command lists files by name. If a directory is named, its contents are listed. If no filenames are given, the contents of the current directory are listed instead. Typing this in after the command prompt:

```
$ ls
```

would produce something like this:

```
dev  ebug.exe  feq.exe  lang  makefile
app  docn
```

Concatenating Files with cat

The `cat` command is used to concatenate the contents of any named files. By typing in this:

```
$ cat
```

it is possible create a new file, from text entered at on the standard input, `stdin`, which is initially the keyboard. Redirection and the concepts of standard input and output make `cat` more powerful than it might at first appear.

When a filename is specified, like so:

```
$ cat <filename>
```

the file in question is printed to the screen. For example:

```
$ cat demo/example/hello.asm
```

Multiple files can be specified on one line, so the full command syntax is:

```
$ cat [<filename> ...]
```

Changing Directory with `cd`

```
cd [<directory>]
```

This function changes the current working directory to another specified directory. The shell is not currently smart enough to recognise when a name is that of a directory, rather than an executable file, and infer the `cd` from the context, as a Classic Amiga would, so the `cd` command prefix is mandatory.

Printing the Working Directory with `pwd`

```
pwd
```

In case you are not sure where `cd` has got you, `pwd` displays the name of the current directory to standard output.

Creating New Directories with `mkdir`

```
mkdir <pathname>
```

This command creates a directory with the specified pathname. The Classic Amiga equivalent is `MAKEDIR`.

Leaving Amiga with `Exit` or `Shutdown`

`Exit` terminates the shell, while `shutdown` shuts down Amiga. These have almost the same effect from the initial shell, but nowhere else.

It is strongly recommended that this command should be used to shutdown the system, rather than clicking on the cross in the upper left-hand corner of the Amiga dialogue box. Using `Exit` will secure Amiga against data corruption.

Available Editors

You're free to use any editor you like - there are plenty on the Linux side of the development system, including Emacs, the parent of the Classic Amiga's MicroEmacs. This part of the chapter explains a couple of the basic ones that are included among the shell commands, and which you might find useful in scripting or simple configuration.

The Shell Line Editor

The Shell Line Editor (SLE) is a command line editor for the Amiga shell. It consists primarily of shell functions, with a few extras built-in to perform basic operations. The following keys can be used to perform typical operations, although this choice of keymappings is easily re-configurable. The current set makes rather more sense menemonically than it does positionally.

The caret symbol "^" indicates that the control key, sometimes marked CTRL or CTL, must be pressed before and released after the associated letter. This distinguishes the 'control' effect from the literal character.

^A, Home key	Beginning of line
^E, End key	End of line
^B, Left arrow key	Back one character
^F, Right arrow key	Forward one character
^D,	Delete character under cursor
^H, ^?	Delete character to left of cursor
^K,	Delete to end of line
^C,	Delete entire line
^P, Up arrow key	Previous line in history
^N, Down arrow key	Next line in history
Page Up key	Beginning of history
Page Down key	End of history
^Q, ^V	Insert next character literally

The SLE reserves all environment variables whose names start with "sle.".

The default Amiga keymapping is similar to the Emacs keymapping, except that Esc and Del act as they do in the doskey keymapping.

^A Home	Go to beginning of line
^B Left Arrow Key	Go Back A character
^C	Delete whole line

^D	Delete character or end of line
^E End of Line	Go to end of line
^F Right Arrow Key	Go forward a character
^H	Delete previous character
^I	Line Completion
^J ^M	Accept or Newline
^K	Delete to End
^L	Clear Screen
^N Down Arrow Key	Downwards Command History
^P Up Arrow Key	Upwards Command History
^Q	Quoted Insert
^R	Redraw
^U	Delete Whole Line
^V	VI Quoted Insert
Esc	Delete Whole Line
^?	Forward Delete Characters
PgUp	Go To Beginning of Command History
PgDn	Go To End of Command History
anything else	Self Insert

The following command can be used to re-set the default keymapping.

```
set sle.keymap.Amiga
```

A range of key mappings are available within Amiga. Their details of which are provided here for reference. The following command sets the Emacs keymapping as the default.

```
set sle.keymap.emacs
```

Emacs keymapping

^A Home	Go to beginning of line
^B Left	Go back one character
^C	Delete entire line
^D	Delete character or end of line
^E End	Go to end of line
^F Right	Go forward one character
^H ?	Backward delete one character
^I	Line completion
^J ^M	Accept, Enter, Return or new line
^K	Delete to end
^L	Clear screen
^N	Down
^P	Downwards command history
Down	Upwards command history
^Q	Quoted insertion
^R	Redraw
^U	Delete whole line
^V	Vi quoted insertion
PgUp	Display beginning of command history
PgDn	Go to end of command history
anything else	Self insert

```
set sle.keymap.
```

^C	Delete whole line
^H	Backward delete a character
^J ^M	Accept or newline
Esc	Delete whole line
^?	Forward delete characters
Up	Upwards command history
Down	Downwards command history
Left	Go back a character
Right	Go forward a character
End	Go To end of line
Home	Beginning of line
PgUp	Beginning of history
PgDn	End of history
anything else	Self insert

Key Bindings

These bindings are merely the default. It is possible to reconfigure the Amiga Shell so as to obtain a key binding for any desired type of text editor, by redefining `sle` keymap within the file:

```
Amiga/home/*/shell.rc.
```

In terms of the overall character set, the Amiga Shell has been configured to emulate the Unix key bindings by default.

See also **parse** and **display** in *The Amiga Shell Commands Reference Manual*.

ED

Synopsis:

```
ed [<filename>]
```

`ed` is the standard text editor. `ed` is a line editor, and reads textual commands from standard input. Consequently it can be used in scripts to automate editing tasks, in addition to being usable interactively - at a pinch.

At all times, `ed` maintains a buffer. When editing a file, the buffer contents are not automatically reflected in the file contents - when editing is complete, the buffer must be explicitly written back to the file.

If a filename is specified on the command line, an ‘`e`’ command with that filename as its argument will be executed before starting to read commands.

Regular Expressions

`ed` understands POSIX Basic Regular Expressions. Regular expressions are always delimited by a specific character, most commonly `/`. The delimiter is not treated as a delimiter if it is escaped by a backslash in the regular expression. Newlines cannot appear in regular expressions; they’re always matched against single lines.

Where a regular expression can legitimately appear at the end of a line, the closing delimiter may be omitted, in which case the closing delimiter is implicitly added, and a ‘`p`’ appended after the delimiter. An empty regular expression is equivalent to the last regular expression encountered.

Addresses

Each address refers to a line in the buffer. There is also a notional 'line zero', which does not correspond to any line in the buffer and is only valid for some commands. There is at all times an address known as the 'current line'.

The following address forms are understood:

The current line

\$ The last line in the buffer (line zero if there are none).

<number> The line with the specified number (which may be zero).

'<letter> The mark referred to by the specified lower case letter. Marks are defined using the 'k' command.

/<BRE>/ The first line matching the specified regular expression. The search starts on the line following the current line, and if it reaches the end of the buffer will wrap back to the first line and continue up to and including the current line. It is an error if no line matches.

?<BRE>? The first line matching the specified regular expression, searching backwards. The search starts on the line preceding the current line, and if it reaches the start of the buffer will wrap back to the last line and continue up to and including the current line. It is an error if no line matches.

[<address>]+[<number>] The specified address, defaulting to the current line, is evaluated, and the specified offset, or one if none is explicit, is added.

[<address>]-[<number>] The specified address, defaulting to the current line, is evaluated, and the specified offset, which defaults to one, is subtracted.

Ranges

Where a command expects a range - two addresses - two addresses may be specified separated by ",", or ";". If separated by a comma, the two addresses are evaluated normally. If separated by a semicolon, the first address is evaluated normally, but the second address is evaluated with the current line temporarily set to the first address.

A range can also be specified as a single address, in which case the range end points are identical. The special range `,"` is shorthand for `"1,$"`, and `;"` similarly stands for `".,$"`.

In any case, to be valid, the end point of a range in the buffer must not precede the start.

Commands

ed commands have a consistent form. There is an optional address or range, followed by a single-letter command, possibly followed by arguments. The command letter, and each part of the address, may be preceded by white space.

After the arguments, most commands may be suffixed by `'l'`, `'n'` or `'p'`. This has the effect of executing the `'l'`, `'n'` or `'p'` command after the main command has completed. These suffixes cannot be used on certain commands, noted below, that would interpret them as arguments.

Each command can be preceded by zero, one or two addresses. It is illegal to give a command more addresses than it wants. Each command that requires addresses has a default which is used if it is given zero addresses, and a single address can always be used as a range in which both addresses are identical. The command synopses below indicate the default address and the number of addresses required for each command. Address zero is invalid, except where noted.

`. a`

Reads lines of text, terminated by either a line containing only `"."` or by the end of input. The text is inserted into the buffer after the addressed line. Address zero is valid, and causes the text to be inserted at the beginning of the buffer. The current line is set to the last inserted line, or the addressed line if there were none.

`., c`

The addressed lines are deleted, and then replaced by text read in the same manner as for the `'a'` command. The current line is set to the last inserted line. If no lines were inserted, the current line is set to the line after the last line deleted; if the lines were deleted from the end of the buffer the current line is set to `"$"`.

`., d`

Deletes the addressed lines from the buffer. The current line is set to the line after the last line deleted; if the lines were deleted from the end of the buffer then the current line is set to `"$"`.

e []

The filename argument, if present, may be preceded by white space, and extends to the end of the line. Suffixes cannot be used.

The entire contents of the buffer is deleted, and then the specified file read in in the manner of the 'r' command. If no filename is specified then the currently remembered filename is used. The currently remembered filename is set to the filename that is used; if a shell escape is used as the filename then no filename is remembered.

The user is protected from destroying a modified buffer with this command in the same way as described for the 'q' command.

E []

This is identical to the 'e' command, except that the user is not protected from destroying the buffer.

f []

The filename argument, if present, may be preceded by white space, and extends to the end of the line. Suffixes cannot be used. If a filename is specified, the currently remembered filename is set to the specified filename. Whether the name is changed or not, the currently remembered filename is then displayed on standard output.

h

A help message is displayed on standard output, explaining the last error that occurred.

H

Toggles a mode, initially off, in which a help message is displayed for each error that occurs, immediately after the "?" notification. If turning the mode on, and an error has already occurred, it is explained in the manner of the 'h' command.

. i

Reads text in the same manner as the 'a' command, inserting it before the addressed line. The current line is set to the last inserted line, or the addressed line if there were none.

..**j**

If only one line is addressed, this does nothing. Otherwise, it joins the addressed lines together, removing intermediate newlines, and sets the current line to the joined line.

..**k**

Sets the specified mark to the addressed line. There are 26 marks, referred to by lower case letters. Marks remain attached to the same line regardless of how that line moves.

..**l**

Writes the addressed lines to standard output in a visually unambiguous form.

Unprintable characters and backslashes are represented in the backslash escape form used in the C programming language; long lines are split with a backslash newline sequence; and each line is terminated by a "\$". This is identical to the output form of sed's 'l' command. The current line is set to the last line displayed.

..**n**

The addressed lines are written to standard output, each preceded by its line number and a tab character. The current line is set to the last line displayed.

..**p**

The addressed lines are written to standard output. The current line is set to the last line displayed.

P

Toggles the display of a prompt when reading commands. The default prompt is "*". By default, the prompt is disabled. The -p option sets the prompt string and causes the prompt display to be initially enabled.

q

Causes ed to exit. End of file is also treated as a 'q' command.

If the buffer contents has been modified since the last 'e' command or 'w' command that wrote the entire buffer to a file, it is an error. But if the 'q' command is then repeated, with no intervening commands, it executes normally.

Q

Causes ed to exit. This is identical to the 'q' command, except that the user is not protected from destroying a modified buffer.

\$ r []

The filename argument, if present, may be preceded by white space, and extends to the end of the line. Suffixes cannot be used. The specified file is read (by default the currently remembered filename). If the filename given begins with "!", the rest of the line is taken as a shell command which is run, and its output is read instead.

If the last byte read is not a newline, then a newline is silently appended. The number of bytes read is written to standard output. The data read is inserted into the buffer after the addressed line. Address zero is valid, and causes the data to be inserted at the beginning of the buffer. The current line is set to the last inserted line, or the addressed line if there were none. If there is no currently remembered filename, and a filename not beginning with "!" is specified, then this filename becomes the currently remembered filename.

Search and replace

..s/ <BRE>/ <replacement>/ <flags>

Any character may be used to delimit the regular expression and replacement string. A slash "/" is used in the example. The closing delimiter of the replacement string may be omitted, in which case a 'p' suffix will be implicitly appended to the command.

On each of the addressed lines, this searches for the specified regular expression, and replaces the first match with the specified replacement string. The current line is set to the last line on which a substitution occurred; it is an error if no substitutions occur.

In the replacement string, "&" is replaced by the portion of the line that matched the regular expression. "\1", "\2" and so forth are replaced by the corresponding matching sub-expression. Any character other than digits can be escaped by preceding it with a backslash.

Search and replace flags

g

Substitute all substrings matching the pattern, not just the first.

<number>

Substitute the matching substring, instead of the first.

.., t[<address>]

Inserts a copy of the addressed lines after the specified address (by default the current line). Address zero is valid for the target, and causes the text to be inserted at the beginning of the buffer. It is not permitted for the target to be within the range of copied lines. The current line is set to the last inserted line.

u

Undoes the last modification to the buffer, and sets the current line to what it was before the modification. The 'u' command itself counts as a modification, so repeated use of the 'u' command will flip between two states of the buffer.

Changing marks and the current line does not count as a modification. Conversely, commands capable of modifying the buffer, such as 'a', count as undoable modifications even if they don't actually change the buffer contents.

l,\$ w [<filename>]

The filename argument, if present, may be preceded by whitespace, and extends to the end of the line. Suffixes cannot be used.

The addressed lines are written to the specified file (by default the currently remembered filename). If the filename given begins with "!", the rest of the line is taken as a shell command which is run, and the lines are written to its input. The number of bytes written is written to standard output.

If there is no currently remembered filename, and a filename not beginning with "!" is specified, then this filename becomes the currently remembered filename.

\$ = The line number of the addressed line is written to standard output. Address zero is valid.

!<command>

The command argument extends to the end of the line. Suffixes cannot be used. The specified command is executed. When it completes, a "!" is written to standard output. If the first character of the command is "!", it is replaced by the last shell command used via '!'; thus the ed command "!!" repeats the last '!' command.

..+1

The null command defaults to the 'p' command, but has a different default address and only accepts a single address.

ED Options

-i

Enable interactive mode. By default, ed is in interactive mode if and only if its standard input is a text terminal emulation, known as a tty.

This mode affects the handling of error conditions. Whenever an error occurs, a "?" is displayed on standard output. In non-interactive mode, ed then terminates immediately. In interactive mode, however, only the current command is aborted, and further commands are accepted. In either case, when exiting, ed will exit with a non-zero exit status if and only if at least one error occurred.

-I

Disable interactive mode so that any error terminated ED at once.

-p <string>

Set the prompt string to the specified string, and enable its display. (See the description of the 'P' command.)

-s

By default, the file reading and writing commands report the number of bytes they read or wrote, and the '!' command upon completion outputs a "!" to indicate so. This option suppresses these outputs. This is useful in scripts.

JOVE

Synopsis:

```
jove [-d dir] [-w] [-t tag] [+ [n] file] [-p file] [files] jove -r
```

Description

JOVE is based on the original Emacs editor written by Richard Stallman at MIT. Although JOVE is generally compatible with Emacs, there are differences between the two editors, so you should not assume that their behaviours will be identical.

Invoking Jove

If JOVE is run with no arguments the user will be placed in an empty buffer, called Main. Otherwise, any arguments supplied are considered file names and each is "given" its own buffer. Only the first file is actually read in - reading other files is deferred until you actually try to use the buffers they are attached to.

This is for efficiency's sake; most of the time when JOVE is run on a big list of files, only a few of them actually get edited.

The names of all of the files specified on the command line are saved in a buffer, called **minibuf**. The mini-buffer is a special JOVE buffer that is used when JOVE is prompting for some input to many commands (for example, when JOVE is prompting for a file name).

When the user is being prompted for a file name, they can type Ctrl N and Ctrl P to cycle through the list of files that were specified on the command line. The file name will be inserted where the user is typing, who can then edit it, as if they had typed it in themselves.

Jovial Help

Help about JOVE commands should be accessed via JOVE itself. To access help press Escape followed by ?. This will bring the 'describe-command' prompt. The user should then enter the command they want help on; it displays information on what keys are set up for that command and a short description of the command.

Jove Options

-d

The following argument is taken to be the name of the current directory. This is for systems that don't have a version of C shell that automatically maintains the CWD environment variable. If -d is not specified on a system without a modified C shell, JOVE will have to figure out the current directory itself, which may be slower than normal. It is possible to simulate the modified C shell, by putting the following lines into the C shell initialisation file:

```
(.cshrc): alias cd 'cd \!*; setenv  
CWD $cwd' alias popd 'popd \!*;  
setenv CWD $cwd' alias pushd 'pushd \!*;setenv CWD $cwd'
```

+n

This option reads the file, as designated by the following argument, and positions the point at the n'th line instead of the (default) 1'st line. This can be specified more than once, although this is unlikely to be necessary. If no numeric argument is given after the +, the point is positioned at the end of the file.

-p

This option parses the error messages in the file designated by the following argument. The error messages are assumed to be in a format similar to the C compiler, LINT, or GREP output.

-t

This option runs the find-tag command on the string of characters, immediately following the -t if there is one (as in -tTagname), or on the following argument (as in -t Tagname) otherwise.

-w

This option divides the window in two. Either the same file is displayed in both windows, or the second file in the list is read in and displayed in its window.

Reconfiguring Shell Interaction

Shell Interaction

The Amiga Shell reads input from a source which can be selected as described in the Options section below. If invoked with no arguments or options, it executes the interact command. For further information upon this command please see *The Amiga Shell Commands Reference Manual*.

The input is interpreted in accordance with the grammar presented below. The shell exits with the status of the last command it executed, or zero if it didn't have any commands to execute.

Shell command synopsis

```
shell [<arg> ...]
```

This runs a shell. Shells may be run recursively to a number of levels limited only by available memory.

Shell command options

-c <command>

This function will only execute the specified command, in the manner of the `eval` command (please see list of shell commands for further information). This takes precedence over `-s`.

-s

Read commands from standard input. By default, commands are read from standard input if there are no command line arguments. If arguments are given, and this option is not used, the first argument is taken as a filename, and commands are read from that file, in the manner of the source command.

-I

This function commands the shell to cease being interactive. By default, the shell is interactive if it is reading commands from standard input. If this is not the case the shell cannot be interactive.

An interactive shell will prompt for each line of input. It will execute each complete command as soon as it is read, rather than parsing the entire input before doing anything, and on a parse error or other meta-error will scrap only the current line and will prompt for a new command.

Advanced Shell Usage

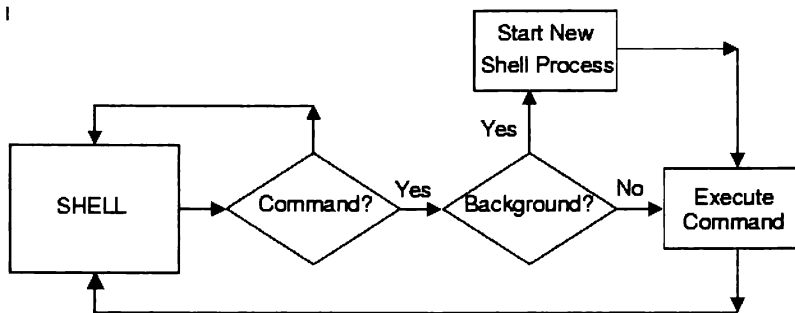
Multitasking Functions

In certain circumstances a command may be run in a subshell, if for instance, it is necessary to "background" a process. Using the **sub** function means that the command is run in a separate process from the current shell environment. Redirections and shell variable settings in the subshell cannot affect the main shell. This may justify using the **sub** command. The syntax for this is:

```
sub <command>
```

If backgrounding of a process is required, then the **-b** option should be specified on the command line. Use of this option causes the shell to not wait for the subprocess to terminate before returning. Sub's exit status is zero. Exceptions thrown in a subshell cannot be caught in the invoking shell process, and the subshell's exit status is truncated to the size of a process exit status. The shell variable `shell.pid` always refers to the main shell's PID, even in a subshell.

The diagram illustrates the logical flow of the shell background process.



Process status

The `ps` command provides information about all processes being run by Amiga.

```
ps [<pid-list> ...]
```

This function displays the status of processes. The requisite processes may be specified by their process ID, but otherwise the status of all processes will be displayed.

Running Shell Scripts

To create a shell script type in the following:

```
$ cat > myscript.scr
ls
echo myscript.scr executed
Ctrl-D
```

To run this script enter either of the following commands at the shell prompt:

```
$ source myscript.scr

$ shell myscript.scr
```

By default script files are stored wherever they are created. Script files can contain any number of shell commands lines. Each line is executed sequentially, unless specified otherwise.

Shell Variables

Like most programming languages the shell allows you to define variables, and can then keep track of an arbitrary number of them. The variables are local to the shell process, but are exported to the environment, which is itself a set of variables which all commands run have access to. Variables are initialised from the shell's environment.

The working environment is defined at login, and is set by using the values that the shell reads as it starts up. You can change your working environment by editing these files and setting new values for their variables. This can be done by using the set command, which is used to set the value of the specified shell variable to the array consisting of any arguments that may have been specified.

```
set <var> [ <arg> ... ]
```

For example:

```
$ set myvar /home/ hello
```

After that type in:

```
$ ls $myvar
```


This turns the shell prompt into a part of the variable name. Further information about this function can be found in *The Amiga Shell Commands Reference Manual*.

All shell variables are one-dimensional arrays of strings. The empty array is permitted, and is distinct from the array containing only the empty string. These are both, in turn, distinct from an unset variable, which has no value.

In the case of a variable whose value is the empty array, or has more than one element, the corresponding environment variable is only an approximation of the true value of the variable. However, the true array value is actually encoded in another environment variable, with a special name. This means that programs that know about this mechanism, such as the shell itself, can inherit array variable values.

Variable names

The shell permits any string to be used as a variable name. By convention, variable names are hierarchical; a prefix indicates which part of the system uses a variable.

However, the shell reserves all variables whose names start with "shell.". Such variables may be modified at any time by the shell, and modifying them may have unexpected consequences.

The following shell variables have special meanings:

shell.

This prefixed is reserved for the shell itself.

shell.argv

This is initialised to the shell's command line arguments, possibly after one has been taken as a script file name.

shell.func.<func>

Stores the definition of the shell function <func>.

shell.argv0

This is initialised to the shell's argv[0]. If a shell script file was specified on the shell's command line, its name is used instead.

shell.binpath

An array of directories to search to find tools to execute as builtin commands. If it is not set in the environment, the shell will initialise it to a suitable value to make it possible to execute the "set" command.

shell.path

An array of directories to search for external command scripts and tools.

shell.pid

Initialised to the process ID of the shell.

shell.ppid

Initialised to the parent process ID of the shell.

shell.spath

An array of tools to execute to try executing simple commands. Each tool on this path gets an attempt at each command, until one of them declares that it has succeeded in executing the command. If it is not set in the environment, the shell will initialise it to a suitable value to make it possible to execute builtin commands. This at least makes it possible to modify this variable.

sys.

This prefix is reserved for system library use.

sys.cwd

The system uses the value of this environment variable as the "current directory" of each process. The shell itself does not treat this specially, but will be so affected by changes in its value.

user.

This prefix is reserved for user configuration parameters.

user.locale

The user's locale, used to determine how dates should be printed, the language messages should be displayed in, and so on. If unset, the default locale is used.

user.tmp

The directory where temporary files should be stored (default "/tmp"). If this doesn't exist, or is not a directory, many programs will not function correctly.

user.home

The user's home directory - default "/" - default argument of the `cd` command.

\$ Expansion of Environment Variables

The shell automatically expands environment variables and replaces them with their true value, so that should any word of any command executed, or the filename in any file redirection contain \$ sequences, they will be expanded before the command is actually executed. This can be done two ways.

The simpler form looks like

```
"$myvar"
```

It takes the name of the specified shell variable (in this case `myvar`), and expands to the value of that variable. All normal characters are valid after the \$, so:

```
"$shell.pid"
```

expands to the contents of the environment variable `"shell.pid"`. It is an error for the variable not to exist.

If the variable name cannot be entered in this manner, it must be quoted and then wrapped in braces

```
"${thus}" ,
```

so:

```
"${{}}"
```

expands to the contents of the environment variable with the zero-length name. In the simplest case, the expression contains only a variable name, and the expansion is the variable's value. Unlike the form without braces, it is possible to specify a variable name that contains special characters, by quoting them, like `"${this\!}"`. If the variable name is omitted entirely, `"${}"`, the expression expands to the empty array.

A `${}` expression may be modified by appending a modifier introduced by a `"!"`. Each modifier looks much like a normal command, but modifiers and commands are not interchangeable. Modifiers don't process options. Modifiers can take arguments, separated by white space. Multiple modifiers can be used in one expression, each one modifying the value resulting from the previous modifier.

For example:

```
${foo ! e 3 !q}
```

takes the value of the variable `foo`, filters it through the modifier `e` with an argument of `3`, and then filters the result of that through the modifier `q`.

The most important modifiers are:

- `c` count array length
- `e` select array elements
- `q` quote strings

If a `$` sequence expands to more than one word, everything that precedes and follows it in that word are duplicated, and are prepended and appended to each element of the expansion. If more than one `$` sequence in a single word does this, each one multiplies the number of resulting words.

A `$` sequence expanding to the empty array (not the empty string) causes the word containing it to disappear. Each `$` sequence in a word is only expanded once, no matter how many words result.

Filename Generation (Globbing)

This section describes how the user can match a character or pattern that they have specified against the files in any directory, then making and displaying a list of all the matches. This is called globbing.

Each word of each command executed, and the filename in each file redirection, may contain special globbing characters. The reader should note that if a word contains any of the special characters in the table below, it is subject to filename generation, and will be replaced by a list of all files matching the glob pattern it represents, and should therefore bear this in mind when writing scripts and so on. The special sequences are:

<code>?</code>	match any single character
<code>*</code>	match any sequence of characters
<code>[...]</code>	match any of the enclosed characters
<code>[^...]</code>	match any character not listed
<code>[!...]</code>	same as <code>[^...]</code>
<code>&[...!...!...]</code>	match any of the listed patterns

x'	match zero or one x's
x'	match zero or more x's
x"	match one or more x's

The globbing characters are used in conjunction with the standard shell functions:

```
ls -l *.txt
```

would therefore produce a listing of all files bearing the suffix `.txt`. All normal characters, and all quoted characters, stand for themselves. All characters that result from \$ expansion are regarded as quoted for this purpose.

Inside a [...] list, a range of characters can be given, as with regular expressions. "[a-e]" is thus equivalent to "[abcde]". For example:

```
ls -l [a-f]*.*
```

would match the listed characters in any files, while this:

```
ls -l [^acdef]*.*
```

would match any character not listed. Any character inside [...] can be quoted using a backslash; this is the preferred way to get a literal "-" into the list. Alternatively, "-" can be placed first or last in the list. "]" can also be included literally by placing it first in the list.

A "/" must be matched explicitly. Pathname components starting with "." can only be matched if the corresponding component of the pattern starts with a literal ".". Pathname components "." and ".." can only be matched exactly.

Pattern Matching

When a shell built-in command or other shell internal provides pattern matching facilities, patterns may be specified as glob patterns. These are defined as in the above section, except that "/" and "." are not treated specially. Also, in most cases all non-glob characters stand for themselves, whereas in normal globbing cases many other special characters are active.

A repeat symbol (' , " or `) can be followed by a ` to make it match the smallest number of times possible, rather than the largest.

Redirection

The characters described below can be used to redirect standard input (`stdin`), standard output (`stdout`), so that the output produced by one program is redirected to another file.

For example:

```
ls > demo/example/test.txt
```

Rather than printing to the screen it prints to `demo/example/test.txt` file. In this instance `demo/example/test.txt` is standard output for the `ls` process. It is also possible to redirect input so that whereas a program would otherwise obtain input from `stdin`, it can acquire it from `demo/example/in.dat` instead.

The symbols used for redirection are identical to those in the Classic Amiga shell:

> Redirection of output

>> Output appended to the end of the specified file rather than overwriting it.

For instance:

```
$ ls >> test.txt
```

will produce a listing within `test.txt` appended to the end of the file.

< Redirection of input

<> Redirection of both input and output

The output redirections will automatically create the file if it does not already exist. If the argument specified expands to multiple words, it is equivalent to specifying a separate redirection for each word, in sequence.

Where multiple redirections appear in a sequence, they are processed in order. A duplication duplicates the state of the file descriptor at that point in the sequence.

The first redirection on any file descriptor replaces the former disposition of that file descriptor. Subsequent non-close redirections on the same file descriptor add to that file descriptor; multiple output redirections are implicitly "tee'd", copied from standard input to standard output; multiple input redirections are implicitly concatenated. Multiple bidirectional redirections, or multiple redirections of incompatible modes, are an error.

Piping

Piping is a way for two processes to communicate with each other. Instead of redirecting to a file it is possible to pipe to another process. The '|' character redirects the stdout of the command on its left to the stdin of the command on its right. For instance,

```
dir | sort
```

pipes output to another program. The basic variants upon this are as follows:

Redirect stdout to a file:

```
x >file
```

Redirect stdout appending to a file:

```
x >>file
```

Redirect stderr to a file:

```
x >(2)file
```

Redirect both stdout and stderr to the same file:

```
x >file >(2)=1
```

Pipe stdout to the stdin of the command y:

```
x | y
```

Pipe stderr to stdin:

```
x | (2>0) y
```

The capability is actually much wider than the standard piping syntax. In particular the shell also enables the user to use zsh-style multios. For instance:

```
ls >x >y
```

copies output to more than one place, and

```
tr a-z A-Z <x <y
```

concatenates input from more than one place. The `tr` command copies standard input to standard output, modifying the data as specified by any options and arguments. For more information upon this command please see *The Amiga Shell Commands Reference Manual*. Combining these capabilities,

```
ls | < x tr a-z A-Z
```

takes input both from a command and a file (concatenating). However,

```
ls > x | tr a-z A-Z
```

will not copy output to both a file and a command, because pipelines associate left-to-right.

```
tr a-z A-Z | (<) ls > x
```

will copy output to a command and a file, or it can be more clearly expressed as

```
ls >x %>{tr a-z A-Z}
```

Exceptions

In order to avoid a plethora of special case checks the shell supports an exception mechanism. An exception is identified by an arbitrary string, determined when the exception is thrown. Unless caught, an exception will abort enclosing shell constructs, and ultimately terminate the shell.

Exceptions can be generated by error conditions detected by the shell, or by user commands. As exceptions are purely a shell concept, external commands cannot generate exceptions - only things done within the shell, including builtin commands and shell functions, can.

Exceptions conventionally start with a hierarchical identifier, in the manner of environment variable names. This makes it easy to use pattern matching to classify an exception with any desired granularity. Following the hierarchical part, exceptions may contain an error message, separated by a "!". If an uncaught exception causes the shell to terminate, this error message is displayed. If the exception does not have such an error message, a standard message is used instead.

These are the exceptions used by the shell:

shell	All shell-generated exceptions have this prefix
shell.err	Error detected by the shell
shell.err.emptycmd	An empty simple command was executed
shell.err.ext	Error creating process for external command
shell.err.file	Error reading file
shell.err.glob	Globbering error
shell.err.mod	Error executing expansion modifier
shell.err.nocmd	A non-existent command was executed
shell.err.novar	A non-existent variable was referenced
shell.err.parse	Parse error
shell.err.redir	Redirection error
shell.err.sub.	Error creating process for subshell
shell.err.usage	Bad usage of a builtin command
shell.exit	Shell exiting
shell.return	Returning from a shell function
shell.return.g	Return from the innermost function
shell.return.s.<func>	Return from function <func>
shell.return.x.<except>	Return and propagate exception <except>

Exceptions can be manually thrown using the built-in command `throw`, and caught using the shell function `catch`. For further information about these please see the *Shell Commands Reference Manual*.

Exception throwing

```
throw <exception> [<status>]
```

This throws the specified exception. The specified numerical status (default zero) is used for this.

Exception catching

```
catch <try-command> <exception-variable> <status- variable> <catch-command>
```

This function executes the <try-command>, and saves its exit status in the variable <status-variable>. If it has exited normally, set <exception-variable> to an empty array, and return that status. If it exited with an exception, save the exception in <exception-variable>, and execute the <catch-command>.

Normally the <catch-command> should rethrow the caught exception if it is of no interest.

Under normal circumstances, one does not wish to catch absolutely all exceptions, and must therefore be careful to rethrow a caught exception. Typically the code to do so looks something like this:

```
catch {  
  ... # code that might throw an exception  
}  
except status {(Note 1)  
case -- $except ~ (Note 2)  
  {...} {  
    ... # handle an exception of interest  
  } ~ ...  
  {*} { (Note 3)  
    throw -- $except $status (Note 4)  
  }  
}
```

The comments in the listing refer to the following notes.

Notes on Catching

Note 1. This should not be represented as \$except or \$status - these are arguments to the catch command, telling it the names of the variables in which to store the exception and exit status.

Note 2. The tilde here prevents the following newline from terminating the command.

Note 3. The asterisk here is a pattern matching anything - the default case.

Note 4. This default exception handler is used to rethrow the code if required.

Shell Grammar Reference Guide

EBNF syntax

This section describes the basic syntax of the shell programming language. It is laid out in the form of BNF syntax, recursively with each following line defining its predecessor. However two extensions to common Backus Nauer Form syntax are used:

```
FOO-seq : FOO
"      | FOO-seq FOO
FOO-opt : <empty> FOO
```

In the first of these FOO-seq may stand either for a single command, or for a sequence of commands. FOO-opt represents optional commands, which may not be specified, or may be specified as an option to another command.

Command lists

```
CMD-LIST  : COMMAND-1-seq-opt      COMMAND
COMMAND-1 : COMMAND TERM
COMMAND   : LWSP-opt CMD
TERM      : <newline> | ";"
```

Here a command list is at least one command, or a list of commands. The input must be a sequence of COMMANDs, separated by TERMS. Please note that TERM is a separator, not a terminator; the final command is delimited by the end of input. In turn a command must be followed by a term, so that a command list could either be one command or a series of commands either separated by ; symbols or by newlines.

A command is a command followed by an optional white space followed by a command. See below for more information about white space characters.

Commands

```
CMD : <empty> | PIPELINE
```

The empty command does nothing; its sole effect is to permit multiple adjacent terminators. The pipeline is the usual form of command. A pipeline is either a simple command such as `ls` or `dir`, or an input redirection.

```
PIPELINE : SIMPLE-CMD
| PIPELINE PIPE-REDIR SIMPLE-CMD
```

The pipeline really runs its final simple command; the "pipeline pipe-redir" part acts like an command redirection for the simple command.

```
PIPE-REDIR : " | " PIPE-SPECS-opt WSP-opt
```

The `| PIPELINE PIPE-REDIR SIMPLE-CMD` e pipe-redir is syntactic sugar for a command redirection. "`PIPELINE | CMD`" is equivalent to "`%<{PIPELINE} CMD`" (see below). If a PIPE-SPECS is specified, it overrides the default. The usual syntax is reversed; "`PIPELINE |(2>0) CMD`" translates to "`% (0<2){PIPELINE} CMD`".

Simple Commands

```
SIMPLE-CMD : PRE-REDIR-opt ARG CMD-PART-seq-opt LWSP-opt
| REDIR-LWSP-seq
PRE-REDIR : REDIR-LWSP-seq-opt REDIR LWSP
CMD-PART : LWSP-opt REDIR | LWSP ARG
REDIR-LWSP : REDIR LWSP-opt
```

A simple command consists of a possibly empty sequence of redirections, and a non-empty sequence of arguments, intermixed freely. See below for further information on arguments. When a simple command is run, it does not terminate until not only the main command process, but also the processes associated with any command redirections, including pipelines have also terminated. The exit status of the whole command is the bitwise OR of the exit status of all these processes.

If no arguments are listed - only redirections - the redirections are applied to the shell itself, affecting all future commands. In this case, the processes associated with command redirections will not be not waited for.

Redirections

```
REDIR : REDIR-OP FD-opt LWSP-opt REDIR-ARG
| CMD-REDIR
REDIR-OP : ">"
| ">>"
| "<"
| "<>"
```

```

FD : "(" DIGIT-seq-opt ")"
REDIR-ARG : ARG
| "=" LWSP-opt DIGIT-seq
| "-" LWSP-opt "-"

```

The "REDIR-OP" indicates the mode of the file descriptor affected, and determines which file descriptor is affected by default. It determines where output is redirected to, and how. This works as follows:

```

> 1 Output
>> 1 Output (append)*
< 0 Input
<> 0 Input and output

```

>> will append to the end of the file rather than overwrite that file.

The output redirections will create the file if it doesn't exist. If the ARG specified expands to multiple words, it is equivalent to specifying a separate redirection for each word, in sequence. If the ARG expands to no words, the redirection is ignored.

If "=" and a file descriptor are specified instead of a filename, the specified file descriptor is duplicated. A "-" means to close the file descriptor being modified.

Where multiple redirections appear in a sequence, they are processed in order. A duplication duplicates the state of the file descriptor at that point in the sequence. The first redirection on any file descriptor replaces the former disposition of that file descriptor. Subsequent non-close redirections on the same file descriptor add to that file descriptor; multiple output redirections are implicitly tee'd, multiple input redirections are implicitly cat'd. Multiple bidirectional redirections, or multiple redirections of incompatible modes, are an error.

```

CMD-REDIR      : "%" PIPE-SPECS-1 LWSP-opt
PIPE-SPECS-1   : PIPE-OP : PIPE-OP
| PIPE-SPECS
PIPE-SPECS : "(" ")"
| "(" PIPE-SPEC PIPE-SPEC-1-seq-opt ")"
PIPE-SPEC-1   : ";" PIPE-SPEC
PIPE-SPEC     : DIGIT-seq PIPE-OP DIGIT-seq
| "<" | ">"
PIPE-OP       : "<" | ">" | "<>"

```

The CMD-LIST is executed in parallel with the main command. Pipes are constructed between these two processes. The CMD-REDIR construct acts like a sequence of redirections to the main process' end of these pipes.

Each PIPE-SPEC specifies a pipe; it contains the main process' file descriptor, the direction of the pipe, and the subordinate process' file descriptor, in that order. "<" indicates a pipe directed from the subordinate process to the main process - an input redirection - while ">" is the reverse, and "<>" is a bidirectional pipe.

The commonest PIPE-SPECs can be abbreviated; "<" means "0<1", and ">" means "1>0". The entire PIPE-SPECS list can be similarly abbreviated: "<" means "(0<1)" ("(<)"), ">" means "(1>0)" ("(>)"), and "<>" means "(0<1;1>0)" ("(<;>)"). It is an error to mention a file descriptor on either side more than once in the PIPE-SPECS.

Arguments

```
ARG : ARG-1-seq
ARG-1 : ARG-NORM | GLOB-PART
ARG-NORM : NORM-CHAR | QUOTE | EXPANSION
QUOTE : "\"" <any char except NUL>
        | BRACE-QUOTE
BRACE-PART : QUOTE | <any char but NUL, "\", "{" or ">">
```

Arguments can contain quoting, globbing and various forms of expansion. In a BRACE-QUOTE, only the outer braces are stripped off: everything inside them is a quoted part of the argument. This can be used to quote an arbitrarily complex set of shell commands as a single argument to a command.

```
EXPANSION : "$" NORM-CHAR-seq
| "$" "{" LWSP-opt NAME-CHAR-seq-opt LWSP-opt MODIFIER-seq-opt" }"
NAME-CHAR : NORM-CHAR | QUOTE
MODIFIER : "!" LWSP-opt NORM-ARG MOD-ARG-seq-opt LWSP-opt
MOD-ARG : LWSP NORM-ARG
NORM-ARG : ARG-NORM-seq
```

This syntax provides access to a wide range of forms of expansion. See the section "\$ Expansion" earlier for details.

```

GLOB-PART : GLOB-SPEC GLOB-REP-opt
| ARG-NORM GLOB-REP
GLOB-SPEC : "*" | "?"

| "[" RANGE-NEG-opt "]"-opt RANGE-CHAR-seq "]"
| "&" "[" ARG-1-seq-opt "]"
RANGE-NEG : "^" | "!"
RANGE-CHAR : NORM-CHAR
| "\" <any char except NUL>
GLOB-REP : "\"" | "'" | ""

```

Globbering is described earlier, in the section headed "Filename Generation".

White space

```

LWSP-2 : <white space characters except newline>
| COMMENT
COMMENT : "#" COMMENT-PART-seq-opt (see note below)
COMMENT-PART : QUOTE |
LWSP-1 : LWSP-2
"~" LWSP-2-seq-opt <newline>:
LWSP : LWSP-1-seq
WSP-1 : LWSP-1 | <newline>
WSP : WSP-1-seq

```

The COMMENT symbol always matches as much input as possible; it can only be followed by a newline, "}", end of input, or an incomplete QUOTE.

"#" starts a comment that continues to the end of the line. The content of comments must have balanced quoting, so that code containing comments will itself have balanced quoting. "~" can be used to continue a command across multiple lines.

Multimedia Toolkit

The Amiga Multimedia Toolkit is a compact toolkit that provides multimedia capabilities. It comprises:

- AVE device driver (`dev/ave`)
- Audio Visual Objects (`ave/avo`)
- AVE toolkit (`ave/avo/*/*`)

The term AVE is frequently used to describe the internal workings of the Amiga multimedia toolkit, as Amiga may be said to provide an Audio Visual Environment (AVE). These two terms are often used interchangeably.

The Amiga multimedia toolkit is supported by digital sound and graphics sub-systems:

- DSFX sound driver (`dev/dsfx`)
- GRF device driver (`dev/grf`)
- SND soundmap class (`ave/avo/snd`)
- PIX pixmap class (`ave/avo/pix`)
- IMG graphics class (`ave/avo/img`)
- Window Management
- Input Methods
- Gadgets
- Event Handling
- TrueType and PostScript Font Engine
- 3D Graphics Driver, providing access to RenderWare
- 3D graphics functionality

The architecture of the Amiga multimedia toolkit gives it great flexibility and allows it to be used on a wide range of devices including games consoles, personal computers, mobile phones, intelligent terminals, network computers, hand-held computers, digital televisions and appliances of all types.

The Amiga multimedia toolkit combines small size with high performance and extensive functionality. It is ideal for applications requiring a multimedia user interface and fast graphics. It is readily extensible, which is an important reason why it has been chosen as the basis for the new Amiga user interface. AVE is still being developed, and will need more work to make it compatible with the Classic model, partly because the old version was so closely tied to obsolete hardware. Work is underway to incorporate the Classic Amiga's BOOPSI object-orientated extensions within AVE, giving it prodigious forward and backward compatibility.

Conceptual Overview

Audio Visual Objects

The fundamental building block of Amiga multimedia toolkit is the Audio Visual Object. This has a class hierarchy, the base class being `ave/avo/class.asm`. This is the from which all Audio Visual Objects inherit. The Audio Visual Objects are:

Application (<i>ave/avo/app</i>)	Gadget (<i>ave/avo/gdg</i>)
Pixelmap (<i>ave/avo/pix</i>)	Soundmap (<i>ave/avo/snd</i>)
Image (<i>ave/avo/img</i>)	Window (<i>ave/avo/win</i>)

These objects may be created, destroyed and organised by the application programmer, as with any other object in the system. Additionally the Audio Visual Objects can be hierarchically organised depending on the needs of the application. Audio Visual Objects can contain other Audio Visual Objects. The 'root' Audio Visual Object is typically an application Audio Visual Object, which any of the other Audio Visual Objects can optionally be added to. In theory, any Audio Visual Object can contain any other Audio Visual Object, with the sole exception of the application Audio Visual Object that serves as a container for all the other Audio Visual Objects.

Audio Visual Objects can be grouped together. Thus a window Audio Visual Object may contain several other Audio Visual Objects, so that when the window is moved the other Audio Visual Objects move too. A typical hierarchy would be an application object containing one or more windows.

Each window contains a mixture of gadgets, pixelmaps and soundmaps. The gadgets may in turn contain pixelmaps or soundmaps, or indeed other gadgets. It is important to note that this hierarchy is entirely flexible and the examples given are arbitrary. The Audio Visual Object hierarchy has a tree structure, with parent Audio Visual Objects having child Audio Visual Objects added to them. The application Audio Visual Object is at the root of the tree.

Each Audio Visual Object provides its own XYZ co-ordinate system for children, and clips children to its bounds. As well as acting as containers, Audio Visual Objects have other characteristics; they have positional data (x and y co-ordinates) and depth (z co-ordinate), they can be placed, moved, resized, hidden and clipped. In addition, each Audio Visual Object possesses an index, which orders it within a z plane, 0 being the foremost.

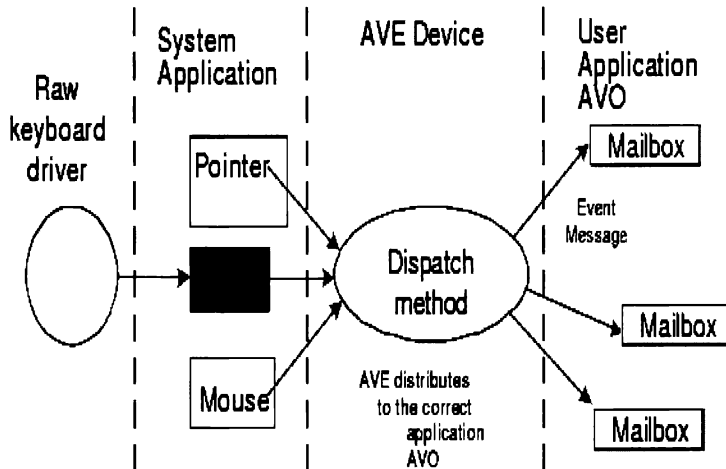
Every Audio Visual Object inherits event handling method code, though it can explicitly override that. An Audio Visual Object can be added to another Audio Visual Object, or easily removed. A number of methods apply to all Audio Visual Objects. Each Audio Visual Object type adds its own additional methods.

The base class code for all Audio Visual Objects is in `ave/avo/class.asm`. All methods of this class are documented in `ave/avo/api.html`. This class is subclassed to produce the other standard Audio Visual Objects, and new ones.

AVE Device Event Handling and Tokens

The primary function of the AVE device is to process incoming events and to pass these on to the correct application Audio Visual Object. The AVE

device contains methods for processing incoming events, namely **dispatch** and **dispatchhk**. The dispatch method handles mouse and pen input events and **dispatchhk** handles incoming keyboard events. The appropriate dispatch method is called directly by system applications handling input from the related device driver. This is illustrated in the diagram.



The generic operation of a dispatch method is:

- Determine the target Audio Visual Object by current token owner or by collision detection. The application object can then be determined by performing a **getparent** method on the target object.
- Create a standard format message including all necessary data, such as x and y co-ordinates, timestamp, target Audio Visual Object, button or key status.
- Send the message to the mailbox of the application Audio Visual Object that was determined in the first step.

The AVE device is also responsible for managing token allocation between Audio Visual Objects. Currently there are four token types defined, though the first two of these are not intended to be used directly.

- **Dragtoken** Gained by an Audio Visual Object that is being dragged
- **Overtoken** Gained by an Audio Visual Object with the pointer over it
- **Keytoken** Claimed by an Audio Visual Object requesting keyboard input
- **Inputtoken** Used by the input method framework

Tokens are used to control access to non-shareable devices. For example, only one Audio Visual Object should have access to a keyboard at any given time, so that would need to obtain the keyboard token before it could accept keystrokes.

The AVE device maintains a list of which Audio Visual Objects have which tokens. Applications can control token handling with these methods:

- **settoken** Set the token's target Audio Visual Object
- **gettoken** Gets the token's target Audio Visual Object
- **clrtoken** Clears any token ownership of the target Audio Visual Object

When the **settoken** method is called the previous token owner is notified with a lost token message, while the new owner is notified with a **gaintoken** message.

AVE Device Calls

These functions associated with the AVE device may be useful to applications:

Tools `dev/ave/tao/lock` and `dev/ave/tao/unlock`

The AVE lock can be used to prevent other processes from affecting the state, for example by issuing an update call. These tools should be used with care, and only around short sections of code. See **`demo/ave/boing`**.

Method `opentoolkit`

This call returns a toolkit instance used to create gadgets.

Method `closetoolkit`

This closes a previously opened toolkit

Method `nextid`

This call returns an AVE unique integer id.

Methods `allocevent` and `freeevent`

These calls allocate and free a standard event message.

Method `aveinfo`

This call returns information such as screen size and mode.

Tool `dev/ave/tao/script`

This processes a script file, starting any applications listed in the file. This tool is used on startup to load the system applications, and by the system menu.

Scripts and the System Menu

Amiga script files list AVE applications with optional parameters. They may contain comments preceded by a hash (#), and use the `.scr` file extension.

The system menu **dev/ave/dsk/runapp** is invoked by clicking on the Amiga backdrop. This makes the directory tree rooted in `/app/start` a menu; each script file found becomes a menu item, while subdirectories become submenus.

To add an option to the system menu, create a script file to invoke an application at the appropriate place in the `app/start` directory tree. The script's filename, minus the extension, becomes the label.

System Applications

The AVE device driver is supported by stand-alone processes. These applications, referred to as **system applications**, handle communications with low level input device drivers and perform tasks such as launching the Amiga multimedia toolkit and providing a tiled backdrop for the main Amiga desktop. These system applications are currently available:

Input device system applications

- Pointer
- Softkey (Virtual keyboard)
- Keyboard

Utility system applications

- Tiles
- Eterm, Amiga Graphical shell
(see `dev/ave/dsk/eterm.html` for more information)
- Launch (start desktop)

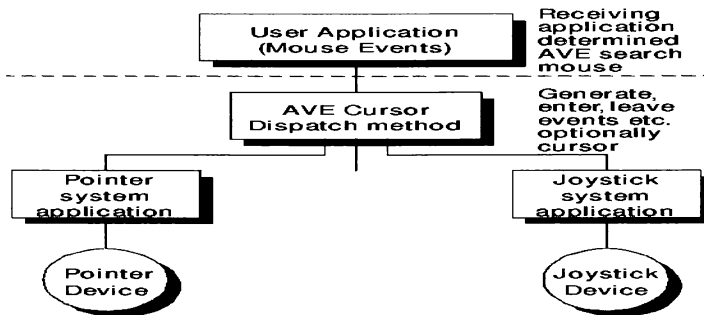
Rather than 'hard-wiring' input device handling and utility code into the main AVE device, a layered approach is taken. With this concept the Amiga multimedia toolkit does not need to know the details of devices that might be connected to it. The low-level details of the device are handled by the appropriate device driver, whilst the system application acts as an interface between that driver and the AVE device. The system application understands the information coming from the device driver and can therefore use the correct dispatch methods to call on the

AVE device. For example, the pointer system application knows how to use the pointer device driver; it knows that that device returns absolute co-ordinates and it knows what information to get from the device driver. It then uses this information to call the dispatch method of the AVE device.

The big advantage of this device concept is that the Amiga multimedia toolkit can quickly and easily be made to support new devices such as pens, mice or joypads. All that is required is a new system application and this is relatively easy to write, assuming that device drivers are available for the devices in question.

Input system applications may be synchronous or asynchronous. The asynchronous type is generally recommended. The synchronous system applications simply poll the input device driver and if any change in state or new data becomes available this is minimally processed before calling the AVE dispatcher. The asynchronous system applications use an asynchronous read method on the input device driver. This **reada** method does not cause the calling application to block. This is based on a callback technique; when the device operation completes or generates some other activity a callback procedure is executed, which then invokes the AVE dispatcher. The callback procedure itself is set up in the system application.

Another feature of the system application architecture is that it is possible to run several system applications feeding into the same dispatch method. For example, it would be possible to have a joystick and pointer all generating input data. The system applications handle this input and then invoke the same dispatch method. From the application's perspective there is only one logical input device, although input may physically come from more than one device. The following diagram illustrates system applications activating the same despatch method on the AVE device:



The system applications can be found in the `dev/ave/dsk` directory.

Audio Visual Object Event Handling

As we have seen, input events from the system applications are filtered by the AVE device. An event format message is created and mailed to the mailbox of the parent application Audio Visual Object of the target Audio Visual Object. This section discusses the Audio Visual Object event handling scheme. It is important to note at the outset that this scheme is defined in the Audio Visual Object base class and is therefore inherited by all Audio Visual Objects.

The Audio Visual Object event handling scheme uses a three layered architecture. Once an event comes into an Audio Visual Object it can be processed by up to three layers of methods, as shown in the table:

1 st level event handler method	2 nd level event handler methods	3 rd level event handler methods
event	systemevent tokenevent mouseevent keyevent commsevent userevent	quitevent gaievent losevent buttondownevent buttonupevent buttonheldevent trackingevent enterevent leaveevent keyupevent keydownevent keycharevent inputevent dropevent n/a

The first level event handler has a switch statement to detect the type of event and then call the correct second level handler method.

The second level handler methods also have a switch statement to detect the type of event and call the appropriate third level event handler method.

By default the third level event handler methods simply return. They should therefore be overridden in order to perform processing of events. It is also possible to override the event handler methods at the first or second level should this be required. For example, if all mouse events must be processed then it may be convenient to override the second level mouse event handler **mouseevent**.

It is also possible to process events without using any of the above event handler methods. To do this a call to `getevent` is made and then the event type determination and handling code is performed within the event loop of the user application. The user application could also call the first level event handler method `'event'` to ensure that other Audio Visual Objects that need to process this event can do so.

Handling Events

Events are received by calling the application's **getevent** method, with an optional timeout. This call sleeps until a valid event is received, then returns the message, target and type of event. The application can either handle the event itself, or pass it to the target. Either way, the event should then be freed by calling the AVE **freeevent** method. Note that a **getevent** timeout returns an event pointer of 0, which must not be freed!

Modality

Mouse and key events are grouped as input events. Within an application, an Audio Visual Object tree can be set to be the modal root, in which case any input events targeted at Audio Visual Objects that are outside the tree are silently ignored. The modal root is normally the application object, in which case all events are accepted.

```
ncall app, setmodal, (app, root_avo:-)
```

sets the modal root to `root_avo`

```
ncall app, setmodal, (app, app:-)
```

resets (clears) the modal root

Standard Audio Visual Objects

The pre-defined Standard Audio Visual Objects inherit from the Audio Visual Object base class. The standard Audio Visual Objects are:

Application	Pixelmap
Gadget	Soundmap
Image	Window

It is possible for user-defined Audio Visual Objects to be created simply by subclassing any of the above Audio Visual Objects. This could be done to create a new type of gadget or window, for example.

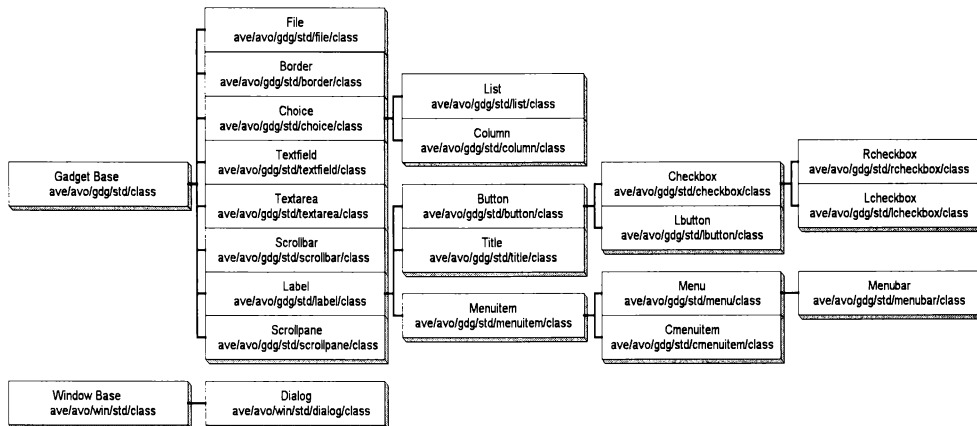
Application Audio Visual Objects

The application Audio Visual Object acts as a container for all other Audio Visual Objects. The application Audio Visual Object is fundamentally an AVE application. AVE applications must create at least one application object in order to use the other Audio Visual Objects. Please remember that the AVE device dispatcher mails the parent application Audio Visual Object of the target object. Therefore there **must** be an application Audio Visual Object to which other objects can be added or removed under program control.

Gadget Audio Visual Objects

The gadget Audio Visual Object base class is defined in `ave/avo/gdg/class.asm`. The interface for the gadget baseclass is defined in `ave/avo/gdg/api.html`. From this gadget baseclass all other gadget Audio Visual Objects are created by inheritance. The subclass gadgets will define their own methods, but inherit all the methods of the base class. Each gadget's Application Program Interface is documented in the file `api.html` in the corresponding gadget subdirectory.

The gadgets created by subclassing the base gadget class are collected into 'toolkits'. A standard toolkit is provided which may be supplemented by additional toolkits to give a different look and feel. Amiga provides techniques for easily switching between toolkits to generate a different look and feel for an application's user interface. This concept of a toolkit also extends to windows. Each subclass whas its own defined interface, documented in the `api.html` file found in the same folder as the subclass. The following diagram further illustrates the toolkit concept.



The gadget subclasses for the standard toolkit (`ave/avo/gdg/std/*`) include:

border	button	checkbox	choice
cmenuitem	column	label	lbutton
lcheckbox	listmenu	menubar	menuitem
rbutton	rcheckbox	scrollbar	textarea
textfield	title	scrollpane	

Window Audio Visual Objects

The window object baseclass is located in `ave/avo/win/class.asm`. Window Audio Visual Object standard toolkit (`ave/avo/win/std/*`) subclasses include dialog. This implements the titlebar, the border, and other controls usually associated with windows, like buttons to minimise or close them.

Pixmap Audio Visual Objects

Pixmap maps are memory areas that contain graphical data. They can be copied to the screen, in which case their contents becomes visible, or they may exist only in memory. Pixmap maps may be created, destroyed, moved, displayed, or manipulated by one of the many methods defined in `ave/avo/pix/api.html`. As pixmap maps are Audio Visual Objects they could be added to other Audio Visual Objects such as a window or gadget. Alternatively, they may just be displayed directly and not added to any other Audio Visual Object. The baseclass pixmap Audio Visual Object is defined in `ave/avo/pix`. The standard pixmap Audio Visual Object is subclassed to provide the following special pixmap types:

1bit 4bit 8bit 12bit 15bit 16bit 24bit 32bit

Pixelmaps are provided to support displays with a wide range of capabilities.

DSFX and GRF

The Amiga system applications use standard device drivers for input from devices such as mouse, keyboard, pen and joystick.

GRF

The Amiga multimedia toolkit calls on the services of the GRF graphics sub-system for graphical output.

DSFX

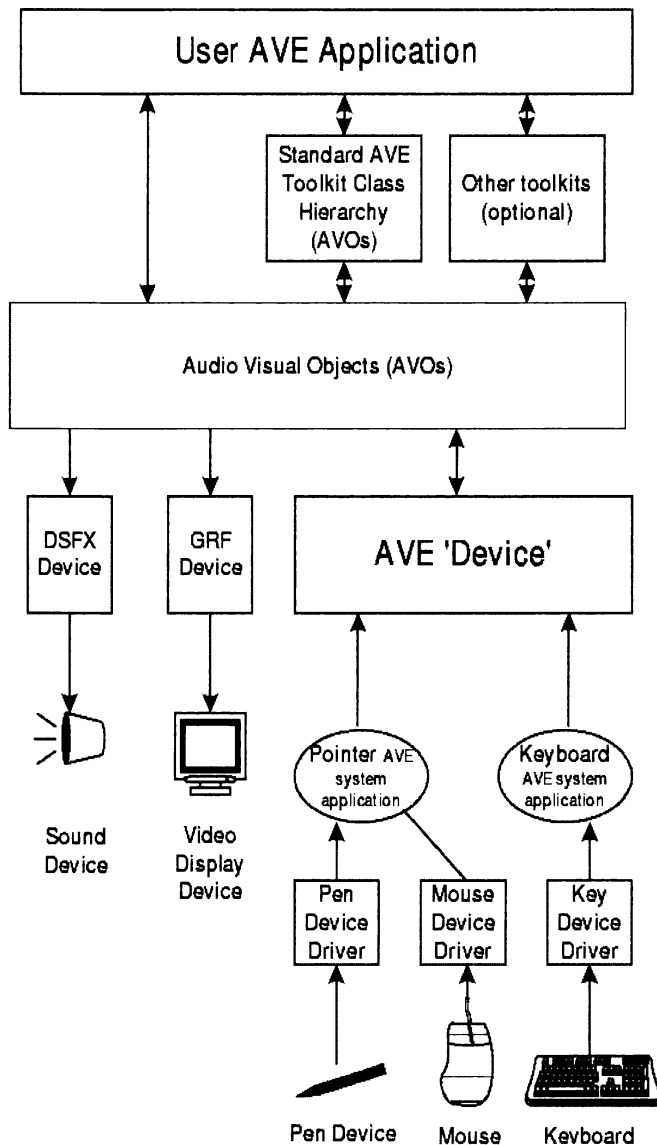
The DSFX digital sound effects (FX) sub-system performs sound output.

Driver files

The device driver component of GRF can be found in `dev/grf` and the driver for DSFX lurks in `dev/dsfx`.

Connections

The relationship between these components is shown in the adjoining diagram.



Directory Structure

The main Amiga components can be found in the `/ave` directory of the Amiga system. The `/ave` directory is divided into a number of sub-directories:

Audio Visual Objects:

<code>/ave/avo</code>	Audio Visual Object base class
<code>/ave/avo/app</code>	application Audio Visual Object class
<code>/ave/avo/gdg</code>	gadget Audio Visual Object class
<code>/ave/avo/pix</code>	pixelmap Audio Visual Object class
<code>/ave/avo/win</code>	window Audio Visual Object class
<code>/ave/avo/snd</code>	sound Audio Visual Object class
<code>/ave/avo/img</code>	image Audio Visual Object class

Standard toolkits:

<code>/ave/avo/gdg/std</code>	standard gadget toolkit
<code>/ave/avo/win/std</code>	standard window toolkit

Other functionality:

<code>/ave/cnv</code>	Pixelmap conversion tools
<code>/ave/flm</code>	Film playing and recording tools
<code>/ave/font</code>	Bitmapped font rendering
<code>/ave/heap</code>	Heap memory management tools
<code>/ave/rec</code>	Region patch tools
<code>/ave/input</code>	Input Method Tools
<code>/ave/ctx</code>	Graphics Contexts
<code>/ave/layout</code>	Gadget Layout Tools
<code>/ave/toolkit</code>	Look and Feel Toolkits

Drivers:

<code>dev/ave</code>	The main AVE driver
<code>dev/dsfx</code>	Sound device driver
<code>dev/grf</code>	Graphics driver

Demonstration Programs:

<code>demo/ave</code>	AVE demo programs
-----------------------	-------------------

Each gadget's Application Program Interface is documented in the file `api.html` in the gadget subdirectory.

AVE Programming

AVE Program Structure

AVE applications have a common structure. As a minimum, a typical Amiga program that uses AVE will need to perform the following steps:

```
Get the AVE device instance
Create application Audio Visual Object
Open toolkit, get properties

Application specific code
Create other Audio Visual Objects as required
Add Objects to their parent Audio Visual Object

Start event loop
    Wait for an event, optionally with a timeout
    process application-specific events
    process quit event EV_QUIT
End event loop

Application specific code

Close toolkit
Close application Audio Visual Object
Exit
```

Blend Example

`demo/ave/blend.asm` is a simple example of programming the AVE toolkit.

Event Types

It is often necessary to detect and process events in the application's main event loop. From the programmer's perspective, the event types are recognised by the following pre-defined constants.

System Events

<code>EV_QUIT</code>	Application has been closed
<code>EV_REFRESH</code>	Screen or toolkit change

Mouse Events

EV_TRACKING	Mouse tracking
EV_BUTTONUP	Mouse button released
EV_BUTTONDOWN	Mouse button pressed
EV_LEAVE	Mouse leaving
EV_ENTER	Mouse entering
EV_BUTTONHELD	Button held past click time

Comms Events

EV_DROP	Drop Event
---------	------------

Token Events

EV_GAINTOKEN	Gained token
EV_LOSTTOKEN	Lost token

Keyboard Events

EV_KEYUP	Raw key up
EV_KEYDOWN	Raw key down
EV_KEYCHAR	Cooked key

User Events

EV_USER	User event
---------	------------

Event Message Format

As ever, the Amiga system is based on message passing. The programmer must be aware of the standard message structure in order for the application to extract necessary information, so this is documented overleaf.

Additional information may be added, dependent on the event type. The relevant `api.html` documents document the precise types of extra information for each type of event.

AVE message structure

This is the AVE message structure:

```
;event message structure
structure MSG_DATA
int32  EVD_TYPE           ;type as per EV equates
int64  EVD_TIMESTAMP      ;time event generated
pointer EVD_TARGET        ;target instance pointer
int32  EVD_TARGETID       ;target object id
pointer EVD_SOURCE        ;source instance pointer
int32  EVD_SOURCEID       ;source object id
int32  EVD_CHANNEL        ;source channel
int32  EVD_X              ;x
int32  EVD_Y              ;y
int32  EVD_RX             ;rx
int32  EVD_RY             ;ry
int32  EVD_BUTTONS        ;button
int32  EVD_COUNT          ;count
int32  EVD_KEY            ;key
int32  EVD_KEYSTATE       ;key state
int32  EVD_KEYCOOKED      ;key cooked
int32  EVD_TOKEN          ;token
struct EVD_DATA, 32       ;data area
```

The file containing this structure is `ave/avo/class.inc`

Event Linking

The Amiga multimedia toolkit allows the creation of links to other Audio Visual Objects. This system allows a number of target Audio Visual Objects to be notified should activity or state changes occur within the source Audio Visual Object. When the source Audio Visual Object generates a state change or action an event is posted into the AVE event handling system. The target Audio Visual Object's main event handling loop will be written to handle the events of interest. This normally involves detecting the event message type and extracting the required information from the message.

Each Audio Visual Object has a number of channels that can be linked to any number of other Audio Visual Objects. Typically Audio Visual Objects have two channels; channel 0 and channel 1. Other channels may be defined depending on the Audio Visual Object.

Channel 0 represents a change state event. This means that if the state of the Audio Visual Object changed, a user event message is sent to the target Audio Visual Object or Audio Visual Objects linked to this channel. The target Audio Visual Objects then decode the message and process the information. Usually the state flag information is appended to the end of the user event message sent. The actual state flags used depend on the type of Audio Visual Object.

Channel 1 represents an action event. When the Audio Visual Object is 'activated', for example by clicking, a user event message is sent to all target Audio Visual Objects linked to channel 1.

Event linking methods

Three methods of the Audio Visual Object base class manage event linking.

Addlink allows a target Audio Visual Object to listen to a particular channel and can be called any number of times to add more Audio Visual Objects to a channel.

It is possible to create event links between an Audio Visual Object and a target Audio Visual Object using the **addlink** method. Each user-defined event link set up by **addlink** needs an event type identifier. This is defined as EV_USER+xx, where xx is any convenient number.

Postlink allows a user defined event message to be sent to Audio Visual Objects linked to a specified channel under program control. This is the method that is

called by the source Audio Visual Object to post user event messages to all target Audio Visual Objects linked to its channels. You may supply a system event type identifier to **postlink** instead of EV_USER+XX.

Sublink removes a link between a channel and a target Audio Visual Object.

An event handler loop

Typically a switch statement in the event handling loop of an Audio Visual Object analyses the event message types and processes the types required. The following code snippet illustrates this:

```
; event handler loop
...
switch
  whencase i0 = EV_BUTTONDOWN
  whencase i0 = EV_TRACKING
  whencase i0 = EV_USER+00
  whencase i0 = EV_USER+01
  ; whencase other events as required
  otherwise
    break
  case i0 = EV_BUTTONDOWN
    ; process this event type
    break

  case i0 = EV_USER+00
    ; process this event type
    break

  case i0 = EV_USER+01
    ; process this event type
    break

  case i0 = EV_TRACKING
    ; process this event type
    break
endswitch
```

A typical example links a dialog window's DIALOG_ACTION channel to an application using EV_QUIT. This means the application will close either because of a system request or a click on the dialog's close button, as illustrated in demo/ave/buttons.asm.

Gadget Programming

Gadget Types

The AVE standard gadget toolkit (`ave/avo/gdg/std`) currently supports these gadget types:

Border	A border
Button	A push button
Checkbox	A push checkbox
Choice	Selected item list
Cmenuitem	A menu item that can be toggled
Column	A column
File	File Browser
Label	Textual label
Lbutton	Latched button
Lcheckbox	Latched check box
List	Scrollable, selectable list
Menu	Pop up menu
Menubar	Pop up menu bar
Menuitem	Item on pop up menu
Rbutton	Radio button (like GadTools)
Rcheckbox	Radio checkbox
Scrollbar	Standard scrollbar
Scrollpane	Scrollable Area
Textarea	Scrollable textual area
Textfield	Editable textual area
Title	A title

The gadget Audio Visual Object class (`ave/avo/gdg/class.asm`) has two main methods besides those inherited from the Audio Visual Object base class.

setstate - this method allows individual state flags to be set, cleared or toggled. The lower 16 bits are reserved for system use, while the top 16 are gadget specific. Any change of state generates a postlink call to channel 0 containing the new state. A mask can be set for the bits that cause an automatic call to update the gadget's appearance. This makes use of the methods **getupdate** and **setupdate**.

getstate - this method returns the current state flags of the gadget.

Each subclass of `ave/avo/gdg/class.asm` may define additional methods. Each gadget's Application Program Interface is documented in the file `api.html` in the gadget subdirectory.

Toolkits and Properties

A toolkit object holds a collection of gadgets with a particular look and feel. Certain aspects of these gadgets, such as background colour, can be further configured via a properties object.

Before an application starts to create gadgets, it should first open a toolkit by calling the AVE method **opentoolkit**. Then it should get a properties object from the application Audio Visual Object, with method **getprop**.

These two objects define which class a particular gadget will use, and what attributes that gadget will have. Gadgets are created by calling **createXXX** on the toolkit, passing in the properties object and any parameters the gadget type XXX requires - see `demo/ave/buttons` for an example.

The properties object reads its attributes from a plain text file, defaulting to `dev/ave/default.prp`. Within this file it is possible to set attributes such as colour for all gadgets, for particular types of gadgets, or for gadgets within a particular application.

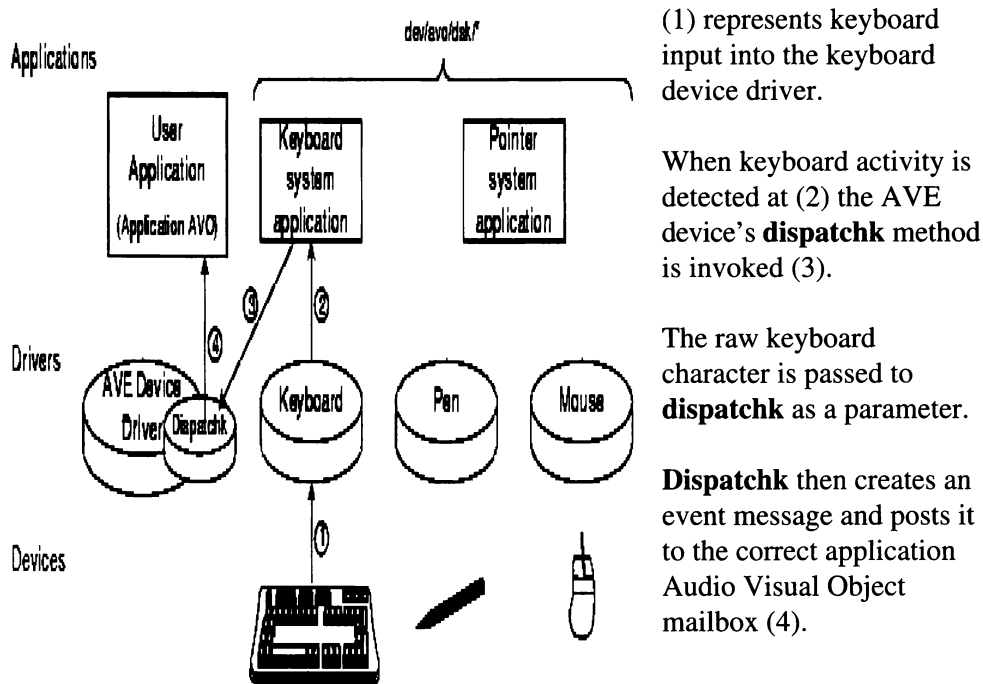
The `default.prp` file contains information on how the system works. The toolkit should be closed when the application exits, using method **closetoolkit** on the AVE.

Anatomy of a System Application

This section looks at the source for a simple system application and highlights the most important features.

The code analysed is for the synchronous keyboard application. This uses a polling loop to check for activity from the keyboard device driver.

The input sequence is illustrated by numbered points in the following diagram:

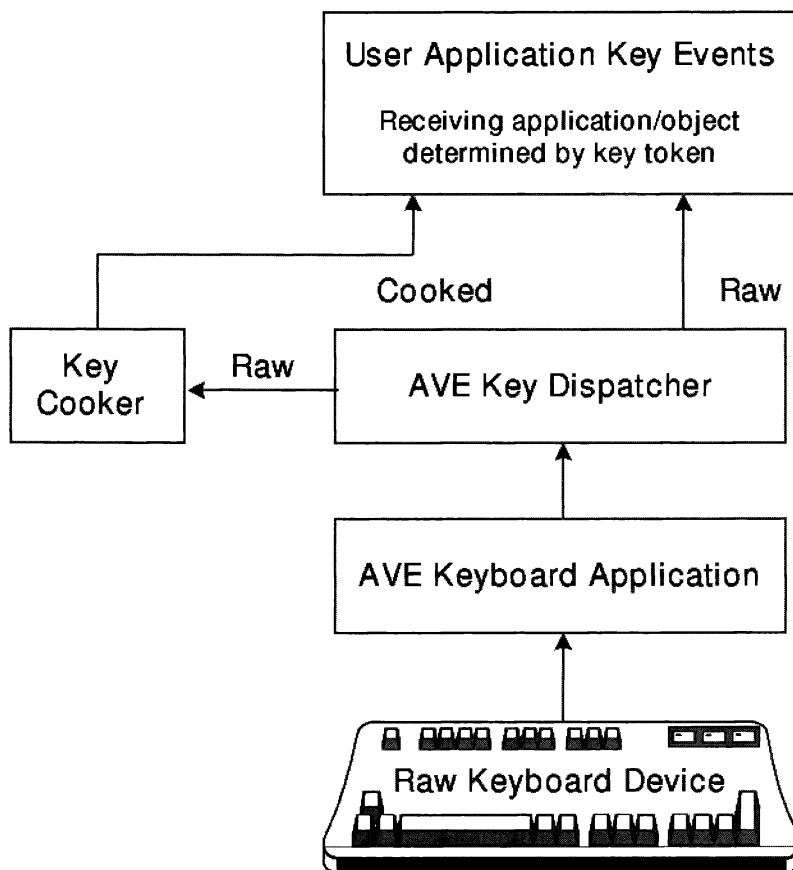


The mailbox would be called a 'message port' in Classic Amiga parlance.

The **dispatchk** method creates an event message that contains both raw and cooked key characters. The terms 'raw' and 'cooked' have their usual Amiga meaning. Raw keys are signals from the hardware which correspond to physical key locations. Once cooked, a keystroke is mapped to a particular character, which might vary in position depending on the system configuration, language and keyboard layout.

Key code conversion

Conversion is achieved by the **dispatchk** method calling the keyboard cooker device to cook the raw key code. The following diagram illustrates the information flow:



Keyboard system code

The Virtual Processor code for the AVE synchronous keyboard system application is listed overleaf:

```

.include 'taort'
.include 'dev/ave/tao/class'
.include 'ave/avo/class'

KEY_LATENCY=CLOCKS_PER_SEC/10
KEY_SIZE=4

structure
int32 KEY_GLB_CODE
pointer KEY_GLB_DEVNAME
size KEY_GLB_SIZE

;option bits
dbitstart
dbit BLOCKING,BBLOCKING

tool 'dev/ave/dsk/skeyboard',VP,F_MAIN,16384,KEY_GLB_SIZE

ent --

defbegin 0
defp argv,ave,app,kdev,khdl,msg
defi evt,flag,key

;set priority to above applications
qcall sys/kn/proc/chpri,(64:i~)

;get args, initialise any instance variable defaults
qcall lib/argcargv,(-:argv,i~)
cpy keyboardname,[gp+KEY_GLB_DEVNAME]

;process any command line options
qcall lib/opts,(argv,options.p,gp:i~,flag)

;lookup ave
qcall sys/kn/dev/lookup,(avename.p:ave,app)
ifnoterrno ave,true
    ;open ave
    ncall ave,open,(ave,app,0,0:app)
    ifnoterrno app,true
        ;lookup keyboard
        cpy [gp+KEY_GLB_DEVNAME],kdev
        qcall sys/kn/dev/lookup,(kdev:kdev,khdl)
        ifnoterrno kdev,true

```

Open keyboard driver for AVE use

```
;open keyboard
if flag?BBLOCKING
    cpy O_EXCL|O_RDONLY,flag
else
    cpy O_EXCL|O_RDONLY|O_NONBLOCK,flag
endif
ncall kdev,open,(kdev,khdl,flag,0:khdl)
ifnoterrno khdl,true
```

Get event from driver

This is the main polling loop:

```
loop
    ;polling loop
    ncall app,getevent,(app,KEY_LATENCY.1:p~,msg,evt)

    ;quit if asked
    if msg!=0
        ;free event
        ncall ave,freeevent,(ave,msg:-)
        breakif evt=EV_QUIT
    endif
```

Synchronous read

Next we perform a synchronous read and invoke the **dispatchk** method with the raw keycode as its input parameter:

```
;call synchronous read to get KEY_SIZE bytes
loop
    ncall kdev,read,(kdev,khdl,gp,KEY_SIZE:flag)
    breakif flag=EAGAIN
    cpy [gp+KEY_GLB_CODE],key
    ncall ave,dispatchk,(ave,key:-)
endloop
endloop

;close keyboard device
ncall kdev,close,(kdev,khdl:i~)
endif
endif
```

```

        ;close ave
        ncall ave,close,(ave,app:i~)
    endif
endif

;close io, return error code
qcall lib/exit,(0:-)
ret

defendnz

setd:
;set device name
ent p0-p2 i0:p0 i0

cpy p1,[p2+KEY_GLB_DEVNAME]
add 4,p0
ret

```

Device data

```

data

options:
;options table for command line processing
dc.i ('d'),setd
dc.i ('b'),OPT_SET+BBLOCKING
dc.i 0

avename:
dc.b '/device/ave/'

appname:
dc.b 'Keyboard',0

keyboardname:
dc.b '/device/keyraw',0

toolend

.end

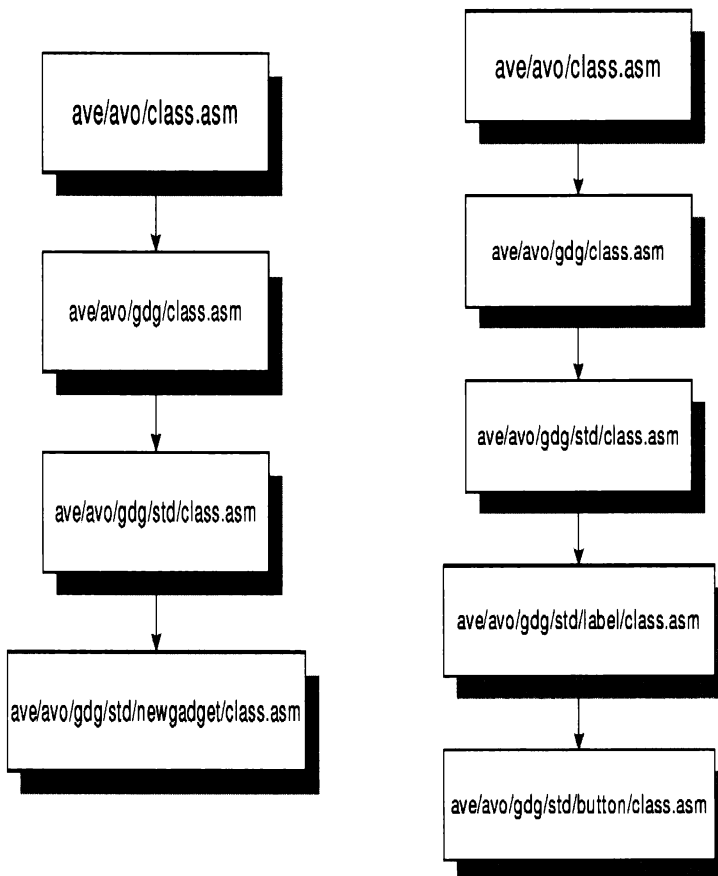
```


Creating New Audio Visual Objects

You must be aware of the hierarchical class structure of the AVE system in order to create new Audio Visual Objects, such as dialog box types or gadgets. The new Amiga system is based on an object-oriented class hierarchy, making full use of inheritance and method overriding concepts.

The task of creating a new Audio Visual Object involves writing a subclass at the appropriate level. This also involves selecting which methods to override, such as third level event handler methods, and adding new methods appropriate to the new Audio Visual Objects features.

The diagram on the left illustrates how a new gadget is created as part of the standard toolkit, deriving properties from a hierarchy of class files. The second diagram shows how the button gadget is developed from the label gadget by two more levels of abstraction:



Displaying Audio Visual Objects

Audio Visual Objects are updated on screen by explicit or implicit update calls. The explicit calls are the Audio Visual Object update and updatepatch methods, while implicit calls stem from certain state changes to gadgets, like gaining focus, or manipulation of Audio Visual Objects, such as calling their change method.

During the update cycle, the redraw method on the avo will be called with an appropriate graphics context. The standard gadget `ave/avo/gdg/std` pre-processes this call, and then calls the `_redraw` method, where the actual gadget drawing occurs. Depending on whether the gadget is buffered or not, it will be passed either a graphics context or a pixelmap as the display surface; although most of the pixelmap interface is reproduced in the context class, the `pick` method does not exist and must not be used.

Audio Visual Objects other than gadgets just overload the redraw method. Pixelmaps call an appropriate blit method, and images typically make use of `modify` or one of the pixelmap drawing methods. Most redraw methods assume they are redrawing the whole of the Audio Visual Object's area; the context handles any clipping.

Patches utilities

The Amiga multimedia toolkit frequently uses rectangular patches, which are small structures allocated from a memory heap object. There is a set of tools in `ave/rec/`, for maintaining lists of mutually exclusive patches. The `api.html` document in the same directory has detailed information about this. These are the available operations: paste a region into a patch list; remove a region from a patch list; copy a region between patch lists; cut a region between patch lists; and clip a patch list to a bounding region.

The routines listed below are used in Audio Visual Object methods to allow regions of the Audio Visual Object to be registered and unregistered for updating:

addpatch (`x,y,width,height`) pastes the region into the update list

subpatch (`x,y,width,height`) removes a region from the update list

updatepatches () updates the regions on the update patch list, then resets the list

Contexts

The Amiga multimedia toolkit uses graphical context objects when updating the screen. These objects support a similar interface to that of pixelmaps, but contain additional information about visible areas.

Normally, an application will never need to use a graphics context directly - they are hidden within the update methods. However the Audio Visual Object redraw methods may be passed in a context as the destination.

Contexts support the read and modify methods, but not the pick method.

Sizing and Layouts

Each Audio Visual Object can return its current, minimum, maximum and preferred sizes. By default, preferred=minimum=current, and maximum is 2^{31} square, giving us more than four million million million pixels to play with.

The size methods can be used in arranging Audio Visual Objects within a containing gadget. To help arrange multiple children, a containing gadget can have a layout object added. The layout object overrides the gadget's getXXXsize methods, by interrogating the children. In addition, there is a layout method that resizes and repositions the children to fit the containing gadget. Layout occurs automatically when the containing gadget changes size.

Two standard layout objects are available - horizontal and vertical. These put all children in one row or column, respectively. These classes reside within the `ave/layout/` directory.

Pix and Img Programming

Pixelmap Audio Visual Objects

Pixelmaps are another subclass of the Audio Visual Object baseclass and inherit all the capabilities of the Audio Visual Object baseclass. Pixelmaps are rectangular areas onto which pixels can be drawn by various methods that are available to the pixelmap class. The pixelmap class methods include those for drawing polygons, circles, lines and pixels.

Colour Encoding

Pixelmap colours are specified internally in a 32-bit ARGB format (alpha, red, green, blue). Alpha refers to the transparency of the colour. Using this technique it is possible to create screens with multiple pixelmaps such that one pixelmap is visible through another pixelmap which has, for example, 50 per cent alpha, making it translucent or half-transparent. 100 per cent is fully transparent.

In certain circumstances pixelmaps may be created that are specified by the programmer with a colour depth that is not comparable with this 32-bit ARGB encoding scheme. In such cases, the internal GRF engine uses an algorithm to translate between the non 32-bit scheme and the internal format.

Alpha

Alpha refers to the transparency of a pixelmap. An alpha of 0% (0) corresponds to an opaque pixelmap. An alpha of 100% (255) means that a pixelmap is entirely transparent. Pixelmaps incorporate an exceptionally fast software alpha-blending algorithm, enabling sophisticated effects without specialised underlying hardware.

Blitting and Copying

The pixelmap class includes methods for copying and blitting (block image transferring) of pixelmaps and pixelmap areas. Blits transfer 2D graphic data from one pixelmap to another. Copies transfer 2D graphic data from one area of a pixelmap to another area of the same pixelmap. Copies are slower than blits as they must handle an overlapping source and destination correctly. Blits can be used to transfer within the same pixelmap, but only if the areas do not overlap.

Drawing Primitives

The pixelmap class also has methods for drawing graphics primitives such as circles, ovals, boxes, ellipses, arcs, polygons, lines and points. In addition, enclosed shapes can be filled in a specified colour as they are drawn. These drawing primitives have been hand-optimised for speed and compactness.

Six combinations exist for the use of any drawing primitive. To be specific, there are three operations, and two filling types. Primitives may be created with a solid fill comprised of one single colour, or they may possess a tiled fill with the pixels being selected from a tile pattern. Following this, they may be alpha-blended with the destination, undergo colour replacement or be XORed.

These drawing methods are documented further in ave/avo/pix/api.html.

Pixelmap Types

The following types of pixelmap are currently supported:

Type	Colour scheme
1	1 bit monochrome
4	4 bit grayscale
12	4:4:4:4 ARGB
15	5:5:5 RGB
16	5:6:5 RGB
24	8:8:8 RGB
32	8:8:8:8 ARGB

Xmethods

Although the `ncall` is generally an efficient way of invoking object method code, an alternative technique is more suitable for situations where it is necessary to execute method code repetitively. An example application is where a large number of pixels need to be plotted sequentially, perhaps to perform a fill operation, unpack a compressed animation, or to render part of an emulator screen held internally in some alien representation.

If a method is likely to be invoked repetitively in this manner it is better to declare it as an **xmethod**:

```
xmethod plot
    ent p0 i0 i1 i2:-
    qcall ave/avo/pix/12bit/12plot, (p0,i0,i1,i2:-),VIRTUAL|FIXUP
    ret
```

This optimisation allows an `ncall` to be made on the method, like this:

```
ncall p0,@plot,(p0:p1)
```

This approach can save a lot of time when repeated calls are performed and the underlying code is relatively trivial. The call overhead might otherwise swamp the time taken to do the actual work.

In this case `p0` is the pixelmap object pointer. `p1` returned by this call is actually a pointer to the method code. Once a pointer to the method code is obtained it is possible to invoke that code using a **gos** instruction. Performing a **gos** to method code is faster than performing a `ncall`. The plot method code could be executed using a call like this:

```
gos p1, (p0, i0, i1, i2)
```

Notice that the parameters passed are now those normally passed to the method.

Xmethods are extensively used in the pixelmap class.

Blitting and Copying

Two method types, **blit** and **copy**, are available for transferring graphics data from one pixelmap to another. Blit and copy methods are similar, with the main difference being that with the blit family, behaviour is undefined when source and destination areas overlap. The copy family correctly handles overlapping source and destination areas, but is somewhat slower than the blit family because of the extra checks carried out. Where overlapping may be involved, then a blit is the required method type; otherwise, a copy will be more efficient.

The alpha variation ensures that the source and destination pixelmaps are blended according to the source alpha value. For the source alpha value 0 is taken as 100% source, implying that the source is opaque, and 255 is taken as 100% destination, meaning the source is completely transparent and so only the destination will be visible. With the XOR version of the method an exclusive-or operation is performed between the source and destination pixels.

Next they are subject to masking and scaling. The masked method of transfer fills the given region with pixels from the source pixelmap. Masked pixels in the source are not written to the destination.

Mixing the trick

It is possible to combine blitting, alpha-blending and masking or copying, XORing and scaling. It is also possible to combine masking and scaling. However, it is not possible to combine alpha-blending and XORing, or blitting and copying operations. This is not a serious loss; if you think about it, those combinations would not make much sense.

Image Audio Visual Objects

Image Audio Visual Objects provide special effects. These are currently available:

- **filled** - plain fill
- **htrans** - fill given colour at half-transparency
- **mshade** - shadow following masked area
- **rgbmask** - logical colour mask
- **table** - table-based colour conversion
- **tiled** - tile given image
- **togrey** - colour to greyscale conversion
- **vgrad** - vertical graduation between colours

A vertically graded image is one where the top and bottom colours are specified, and the colours and translucency get smoothly graduated between these, in the vertical direction. An example of this is `demo/ave/vgrad.asm`.

Sounds

Sounds are implemented by calling on the services provided by the DSFX device driver (`dev/dsfx`) and the associated toolkit (`ave/avo/snd`).

DSFX Device Driver

The DSFX sound driver class provides the following methods:

close - the standard device close method.

getcapability - copies information about the device capabilities into a buffer provided by the caller. The handle may be NULL if information about a particular handle is not required.

getvoice - this method allows applications to check current settings for a voice.

go - this method flushes all current queued commands to the actual hardware.

info - the standard device info method.

log - this method allows an application to transfer a sample to the device and gain a sample handle for future reference. The DSFX drivers expect data to be in the conventional linear PCM sample format.

In other words, samples should be taken at a constant rate with proportional steps in amplitude between consecutive integer values, like the samples from a DAT or CD player. Non-linear step mappings and frequency modulation are not directly supported.

open - open a sound device, like any other.

pause - this method allows an application to pause a voice, allowing playing to be resumed later. The pause commences when a call to the **go** method is made.

play - this method allows an application to prepare to play a logged sample, to specify the asynchronous completion mechanism for playing and obtain a voice handle for the sample. Nothing actually gets played until the **go** method is called.

reference - standard device reference method.

resume - this method allows an application to resume a voice that was paused by calling the pause method. No actual resuming will commence until a call to the **go** method is made. Resuming an ended voice will cause the voice to restart at the beginning of the sample.

setvoice - this method allows an application to change the parameters of a voice that was allocated by the play method. No actual changes take effect until a call to the **go** method is made.

stop - this method allows an application to stop a voice. Unlike **pause**, this method does not allow resumption as it frees any resources allocated for the voice. No actual stopping will take place until a call to the **go** method is made.

unlog - this method allows an application to inform the driver that a previously logged sample is no longer required and that its resources can be freed. Virtual voices allocated by the play method are allocated to real hardware voices during the **go** method call.

The play mode is very useful for checking if a voice has finished playing yet. The sample number lets an application know exactly how far along in a sample the voice has played.

See `dev/dsfx/api.html` for more information.

The Amiga Font System

The Font System consists of three layers: glyph caching technology, vector rendering engine and format decoder plug-ins. Plug-ins support TrueType, PostScript Type 1, Windows Resource Fonts and ROM fonts. Features include single bit, four bit or eight bit anti-aliasing, on-the-fly gamma correction, configurable cache memory, Panose matching, micro-spaced printing, and co-existing plug-ins.

The font manager maintains a list of available fonts and font engines. The manager opens fonts, which are then used to render text. When the font manager is first invoked, it examines the contents of a system file, which contains a listing of the font engines that the system wants to declare as well as the relevant fonts and cache details. This approach means that no font engine is 'hardcoded' into the font system; the required font engines are only defined when the engine is initialised.

The system file `ave/font/store.map` is a plain text file containing the font manager information, as in this example snippet:

```
# Truetype font engine and fonts ...
.engine= ave/font/ttf/    # The TrueType Font Engine
fonts/ANMSSN__.TTF
fonts/ANSNB__.TTF
"fonts/this line has spaces.TTF"
fonts/ANSN____.TTF
fonts/ARCTIC.ttf
# map to file outside of Amiga tree
device/fs/c:/other_os/fonts/times.ttf
```

Font engines are dynamically loaded by the font manager in order to ascertain the validity of the selected font engines. The engines are uniform across the system, since each engine's 'open' tool creates font object classes which inherit from `ave/font/class`. A font object class is used for method calls and for drawing strings.

String drawing is the process of sourcing characters, and linking them together to produce the best typographical representation of the string. The string drawing group of tools provided within the Amiga font system are helper tools, which group together a number of tools and routines into a uniform way of producing text output. At present, these have only a very minor effect on text, but are used to uniformly handle spacing and microspacing across the system.

When creating a tool to use fonts with the Amiga multimedia toolkit, the following methods are available.

ave/font/open Opens a font

ave/font/opens This tool opens a font based on a single string description

ave/font/close Closes an opened font, and frees any associated resources

A string description acceptable to **opens** might be 'Nimrod, 10points' or 'Andale Sans, 12points, italic'

Examples of these methods in action may be found in `demo/ave/font.asm`.

For more information about available methods, consult `ave/font/api.html`

Import/Export Utilities

The `ave/cnv` directory contains tools for import and export of images and sounds. The main tools are `ave/cnv/load` and `ave/cnv/save`.

Loading

The Amiga toolkit can import files of the following types to pixelmaps:

- CPM native compressed pixelmap
- FLM native film (first frame only)
- RPM native raw (compressed) pixelmap
- JPG compressed photographic image
- PCX ZSoft's old Paintbrush format
- BMP format beloved of OS/2 and Windoze
- XBM Unix portable 'X Bit Map' format
- GIF Graphic Interchange File format
- PNG Portable Net Graphic (GIF successor)

For further information about native formats, examine `app/stdio/img2cpm`.

The Amiga toolkit can import files of the following types to soundmaps:

- CSM native compressed soundmap
- RSM native raw (compressed) soundmap
- WAV Windows RIFF-based wave format
- AU Sun's audio sample format

The file type being loaded is determined by the file extension of the filename supplied. This list covers the most common non-Amiga formats; Classic Amiga file formats and anything else you like can easily be implemented as extensions of these sets of types in future. The directory `ave/cnv` contains plug-in routines for file conversion on the fly between alien formats and the new Amiga PixelMap or SoundMap native formats.

Saving

The Amiga toolkit can export files of the following types from pixelmaps:

- CPM native compressed pixelmap
- FLM native film (first frame only)
- RPM native raw (compressed) pixelmap
- TGA Truevision Targa format
- BMP Windows Paint format
- PPM Portable Bit Map colour pixel format

The Amiga toolkit can export the following file types from soundmaps:

- CSM native compressed soundmap
- RSM Amiga native raw (compressed) soundmap

The file type being saved is determined by the file extension of the supplied filename.

There is also a caching loader (`ave/cnv/share`) for images, but not sounds. This is similar to `ave/cnv/load`, except that the bitmap data portion of the pixelmap is cached, so subsequent calls to `share` with the same parameters return a pixelmap pointing to the shared data. Pixelmaps must be of the same type if they are to be shared.

Converting Classic IFF formats

The Amiga system build includes high quality image and audio sample conversion tools, preinstalled on your command path.

Use SOX (sound exchange) to resample and convert 8SVX and other samples; the suite of PBM and PPM conversion tools like `ilbmto ppm`, `pnmtopng` and `pnmtailbm` will convert Classic Amiga ILBM graphics to and from alternative formats via the generic NetPBM ones.

Audio Visual Object bitmasks

AVOs are identified and manipulated within programs using these bitmasks:

Audio Visual Object types :

```
dbitstart
dbit FAVO_APP, BFAVO_APP
dbit FAVO_WIN, BFAVO_WIN
dbit FAVO_GDG, BFAVO_GDG
dbit FAVO_PIX, BFAVO_PIX
dbit FAVO_IMG, BFAVO_IMG
dbit FAVO_SND, BFAVO_SND
```

Audio Visual Object state flags:

```
dbitstart
dbit FAVO_HIDDEN, BFAVO_HIDDEN      Not displayed
dbit FAVO_ALPHA, BFAVO_ALPHA        (pixelmap) alpha blends
dbit FAVO_ROOTPIX, BFAVO_ROOTPIX    Internal
dbit FAVO_SOLID, BFAVO_SOLID        Gadget can be hit
dbit FAVO_TRIG, BFAVO_TRIG          Internal
dbit FAVO_OPAQUE, BFAVO_OPAQUE      Pixelmap has no transparency
dbit FAVO_RECPATCH, BFAVO_RECPATCH Internal
dbit FAVO_PERSIST, BFAVO_PERSIST    Internal
```

Audio Visual Object tokens:

```
dbitstart
dbit FAVO_DRAGTOKEN, BFAVO_DRAGTOKEN
dbit FAVO_OVERTOKEN, BFAVO_OVERTOKEN
dbit FAVO_KEYTOKEN, BFAVO_KEYTOKEN
dbit FAVO_INPUTTOKEN, BFAVO_INPUTTOKEN
```

Audio Visual Object alignment flags:

```
dbitstart
dbit FAVO_ALIGNTOP, BFAVO_ALIGNTOP
dbit FAVO_ALIGNBOTTOM, BFAVO_ALIGNBOTTOM
dbit FAVO_ALIGNLEFT, BFAVO_ALIGNLEFT
dbit FAVO_ALIGNRIGHT, BFAVO_ALIGNRIGHT
dbit FAVO_FILLWIDTH, BFAVO_FILLWIDTH
dbit FAVO_FILLHEIGHT, BFAVO_FILLHEIGHT
FAVO_ALIGNCENTER=0
```

Cast of concepts

AIFF

An acronym for Audio Interchange File Format. A format based on the Electronic Arts IFF scheme extensively used in the Classic Amiga, developed by Apple Computer Inc. for storing high-quality sampled audio and musical instrument information.

Alpha

In this document, alpha refers to the transparency of a pixelmap. An alpha of 0% (0) corresponds to an opaque pixelmap. An alpha of 100% (255) means that a pixelmap is entirely transparent. Not to be confused with DEC's fast processors.

ARGB

The colour model made use of by the Amiga multimedia toolkit. RGB refers to the traditional combination of red, green and blue colours, while A refers to the alpha transparency value.

AU

A sampled audio format developed by Sun Microsystems.

AVE

An acronym for Audio Visual Environment - the domain of the Amiga Multimedia Toolkit.

AVO

An acronym for Audio Visual Object. These are the fundamental building blocks of the Amiga Multimedia Toolkit.

Beans

Reusable self-contained software components, written in the Java language.

Blitting

Block Image Transferring - the process of transferring a 2D region from one memory space to another.

BMP

An exceptionally verbose bitmap file format, developed by Microsoft and also used, in a simpler variation, by OS/2.

Bytecode

A synthetic machine code designed to be concise, general, and easy to generate. Later it may be interpreted or compiled into code for a specific processor.

CISC

Derogatory term used by engineers to classify 'Complex Instruction Set Computers', where the sophistication of some instructions may slow others.

Class

A particular category of software component in an object- orientated system. The properties of an object are determined by its Class. Classes are defined hierarchically, so a each new class is defined by extending an existing one, and may inherit properties from its parents, grandparents and so on. New classes defined in this way are called 'subclasses'.

A Class defines the functionality of objects of that class and the services that such objects provide. A class can be thought of as an object framework. It is possible to instantiate multiple objects of the same class, each object having unique properties.

Classname

The collective name for components of a class. Normally class methods are coded in a file `class.asm`. Each class will therefore have its own subdirectory such as `/path/myclass/class.asm`. In addition to the `class.asm` file there will be a `class.inc` file, which defines the data structure properties for that class, called something like `/path/myclass/class.inc`

Data hiding

The internal functionality within an object is often hidden from the application programmer. Programmer normally manipulates the object through a set of methods. These methods may manipulate the internal data properties of the object. It is safer to manipulate this data through the application program interface as those methods can perform error checking on the request to manipulate properties.

defaultmethod

If the named method called by an application is not present in a class, the defaultmethod will automatically be executed instead. The defaultmethod can code for an error or if coded in a subclass, it can pass responsibility for handling the method to the parent class.

DSFX

An abbreviation for the Amiga multimedia toolkit digital sound effects device driver.

Dynamic Binding

The loading and linking of tools 'just-in-time', or only when they are called for, rather than as part of a monolithic kernel or when programs that might need them later are first loaded.

Elate

Early name for the software foundation upon which the new Amiga is based.

Encapsulation

The process of combining data and access procedures into objects which can be manipulated en masse.

Event

Something that happens which is likely to affect the sequence in which tasks run - typically the arrival of an interrupt or message, or an input action such as the pressing of a mouse button, mouse movement or the pressing or releasing of keys.

Font

A particular style of typeface. A font of a given size is used to perform text string operations like printing and measuring the graphical size of a string.

FPU

An acronym for Floating Point Unit - a hardware sub-system designed to process very large and small values quickly and accurately, usually in IEEE-754 format.

Gadgets

Components provided by the Amiga multimedia toolkit for the construction of a graphical user interface. Menus, windows and scrollbars are all gadgets.

Glyph

Either the outline or bit pattern of a character image, or a surface form derived from a combination of underlying characters in a specific context.

GIF

Acronym for Graphics Interchange Format. A standard for digitised images compressed with the LZW algorithm, GIF was defined in 1987 by CompuServe and updated in 1989, but has declined recently for technical and legal reasons.

GRF

An abbreviated term for the Amiga multimedia toolkit graphics driver.

GUI

An acronym for Graphical User Interface. A graphical environment through which a user may interact with software running a particular device.

IEEE-754

A standard for the representation of fractional values as binary values inside a computer, commonly implemented in FPUs or emulated by software, named after the Institute of Electrical and Electronic Engineers that specified it.

IFF

A standard interchange file format defined by Electronic Arts in 1984 and very extensively used on Classic Amiga systems, as well as in some other applications.

Inheritance

The way that newly defined objects include attributes of previously defined objects. They 'inherit' the properties that have in common with their parent object, reducing the need to copy or rewrite code for every new object.

Every object is or has a baseclass, also known as a parent class. A similar object may reference within its class code, a baseclass thereby inheriting its methods (services), which have already been coded. In this way, subclasses have access to the methods of the baseclass without being aware of their implementation. Each subclass is only aware of its own parent which may itself be a subclass of another class.

Instance

An Object is an instance of a Class. The instance is the data structure that holds the private data of the object. There can be more than one instance of a class in memory at one time with each instance being unique.

JPEG (JPG)

Joint Photographic Experts Group. The original name of the committee that designed the eponymous standard image compression algorithm. Abbreviated to JPG by PPL WHO CNT TYP or WSE PCS ARE BKN.

Kerning

Adjustment of the spacing between characters in groups, thereby improving their appearance.

Latency

The delay between when an activity could start and when it actually does. A realtime system guarantees low latency.

Linux

A freely-distributable operating system modelled on Unix, originally developed by Linus Torvalds of Finland and latterly adopted and adapted by DIY programmers worldwide.

Mailbox

The memory area where messages arrive. On a Classic Amiga this structure would be referred to as a 'message port'.

Method

A method defines a service that an object can perform. Methods can be inherited or overridden by subclasses. This list of methods of a class defines the programmer's interface for that class.

MPEG

An acronym for Moving Picture Experts Group. The term also refers to the family of digital video compression standards and file formats developed by the group.

MP3

The file extension for MPEG, audio layer 3. Layer 3 is one of three coding schemes for the compression of audio signals, designed as an adjunct of the MPEG video standard. It uses perceptual audio coding and psychoacoustic compression to remove less obvious information.

MPEG layers 1 and 2 are public standards, but the layer 3 compression endemic on the Internet is proprietary to German DSP specialists Fraunhofer. Sonically inferior PD layer 3 encoders and decoders are derived from old ISO sources.

NCALL

An `ncall` is a named method call on an object. It is the means by which an object's method code is invoked.

Node

The fundamental unit by which memory is allocated, and the part which objects, messages and other system components have in common. It is the basic building block from which other memory structures are assembled.

Object-based programming

Object-based programming is a synthesis of three concepts: encapsulation, inheritance and polymorphism. Together these ease modular programming.

- Encapsulation permits data hiding by bundling together data and access procedures.
- Inheritance is the ability to include previously defined attributes in your new object.
- Polymorphism is the ability to run code with the same name and interface on different data types.

Objects

Objects provide services and are classified according to the services they provide. An application requesting services has no knowledge of the way in which the object implements those services. An object is an instance of a class. Each object is manipulated via the methods and properties peculiar to that class of object.

PDA

Acronym for Personal Digital Assistant - a small, portable gadget which typically attempts to fill the roles of a toy, a Filofax and a Hitchhiker's Guide to the Galaxy.

PPC

An acronym for Power Personal Computer, a family of microprocessors jointly developed by IBM, Motorola and Apple Computer, based on 1980s RISC designs.

Panose

The PANOSE Typeface Matching System was developed by Benjamin Bauermeister and licensed to Hewlett-Packard Corporation. It's a system for matching static or distortable fonts by objectively classifying them according to their visual characteristics. The name "PANOSE" is derived from the six letters used to distinguish font attributes.

PID

An unique identifying number for a process., allocated locally or by negotiation across a network by the Amiga kernel.

Pixelmap

An AVO defining a 2D graphical region containing graphical information.

PNG

An acronym for Portable Network Graphics. An extensible file format for the lossless, portable and well-compressed storage of raster images, created when patent claims rendered the GIF standard proprietary.

Polymorphism

The facility to use consistent operations on different data types. Addition is polymorphic, in that it may be performed on floating point values or integers of various sizes. The data types may vary but the operation works consistently regardless. The Amiga extends this concept to objects, allowing arbitrary operations to be defined and applied to many types of data, old and new.

PostScript

A scaleable font standard developed by Adobe Systems Inc. PostScript Type One fonts contain hinting information which allows fonts to be rendered more readable at lower resolutions and small type sizes.

QCALL

A clever type of subroutine call which automatically loads the required tool if it is not already in memory.

RISC

Reduced Instruction Set Computing - an attempt to make processors cheaper and faster by concentrating on the simplest, most common operations.

Tao

Literally 'The Way', an holistic philosophy expounded by Chinese philosopher Lao-Tsze some 25 centuries ago - latterly the name of the group of companies that helped us to develop the Virtual Processor and the new Amiga operating system. Please do not attempt to contact Tao for Amiga support - they're busy with other things. Amiga-related questions must be directed to our own support organisation.

TGA

An acronym for Targa Graphics Adaptor. The Truevision Targa Graphics Adaptor file format has outlasted the hardware - the TGA format is a common bitmap file format for storage of 24-bit images.

Thread

A sequence of instructions which is currently executing, and can be performed concurrently with other such sequences

Tool

A re-entrant, re-locatable executable block of code which may be loaded and bound on demand.

TrueType

A font representation derived from the Royal font scheme, initially developed by Apple Computer as a rival to Adobe Systems Inc's PostScript standard. Like PostScript Type 1 fonts, it is an outline font format that allows the scaling of fonts to any size.

Typeface

The features by which a character's design is distinguished from equivalents that look different.

Unicode

A character encoding system designed to support the interchange, processing, and display of written texts in many languages, including Chinese and Japanese. The Unicode standard defines a unique number for every character, symbol and textual element.

Virtual Processor (VP)

A generic computer architecture which encapsulates the main features of modern microprocessors in such a way that programs written for VP can be converted very quickly, as they are loaded, into the native code of any conventional computer.

VLIW

Very Long Instruction Word - a way of designing processors that delivers several instructions simultaneously at each memory access.

VPcode

The instruction set of the Virtual Processor, including symbolic macro instructions.

WAV

An audio format used extensively in Microsoft Windows.

xmethod

A method that allows its method code to be called directly. This is faster than **ncall**, the usual named method call.

XOR

A Boolean exclusive or operation - one where a result is true if one and only one of the inputs is true. One of the set of logical operations codified by George Boole.

Index

▪

- famiga-tool 64
- felate-tool 64
- ftaos-tool 64, 72
- mpackalign 64
- mtrace-all 64
- mtrace-funcs 64, 70
- mtrace-tools 64
- tjcode 44
- Wconversion 66
- Wmissing-prototypes 66

▪

- .setaddr 88
- .setref 88

A

- Abbreviations 31
- Abstract Window Toolkit 36, 44
- abstraction 266
- Access Control 36, 38
- action event 257
- addhead 144-145
- addlink 257
- addnode 144
- addnodeb 144
- addpatch 267
- address error 131
- Address zero 22, 215, 218-219
- addtail 144
- AIFF 279
- AIO structure 188-189
- allocaevent 245
- Alpha 68, 269, 271, 278-279
- alpha-blending 269, 271
- anomalies 171-172
- ANSI C 66, 90, 146, 178, 182, 204
- ANSI library 102
- ANSI-style function 66
- anti-aliasing 274
- applet 53-55
- AppletContext 53
- AppletStub 53

- application Audio Visual Object class 253
- Arabic 56
- Architecture 2-3, 17, 33-34, 36, 45, 48, 128, 147, 161, 170, 177, 195, 241, 247-248, 288
- arcs 269
- ARGB 269-270, 279
- Assembler Language 3, 42, 89, 125
- assembly 23, 26-27, 29, 32, 58, 61, 68, 71, 124, 141, 179
- Asynchronous 42, 164, 168, 184, 188-190, 247, 273
- atom values 9
- attribute 55, 92, 94
- AU 276, 279
- autoincrement 25
- AVE driver 253
- AVE lock 245
- AVE message structure 256
- AVE Programming 254
- AVO 267, 279, 286

B

- backdrop 16, 246
- backing storage 89
- Backus Nauer Form 236
- baseclass 98, 100-111, 250-251, 268, 283
- Bean Information 55
- Beans 50, 54-55, 279
- benchmarks 13, 67, 147
- big-endian 22, 149, 176
- Binding 42, 58, 90, 148, 151-152, 176, 181, 206, 213, 281
- Bit structures 137
- Bitmapped font rendering 253
- Blitting 269, 271, 280
- bloat factor 206
- Block devices 184
- blocking 142, 159, 164, 184, 188, 190, 263
- BMP 276-277, 280
- BNF syntax 236
- BOOL condition 26
- Boolean 31, 288
- boolerrno 143-144, 194
- boolnoterrno 143

BOOPSI 241
border 251, 259
bounding region 267
boxes 7, 16, 46, 197, 269
BPLCON 2
BREAK 6, 26, 32, 39, 99, 138-139, 147, 258
BREAKIF 26, 32, 139, 264
breakiferrno 143
build.target 68-69
button 243, 251, 255, 258-259, 266, 281
byte order 22, 31
bytecode 15, 34-35, 40-41, 45-48, 50-52, 280

C

C Compiler 3, 20, 57, 62-63, 89, 222
caching loader 277
callback 164-165, 189-191, 247
case-sensitive 125
CCALL 29
change method 267
change state event 257
Channel 0 257, 259
Channel 1 257
char 32, 52, 63, 65, 77, 86, 93-95, 101, 239-240
Character devices 185
checkbox 251, 259
child process 155-157, 163
Chinese 287-288
choice 5, 90, 139, 188, 206, 210, 251, 259
circles 268-269
CISC 3, 17, 280
class loader 53-54
class methods 37-38, 99, 268, 280
class variable 37
classend 99-100, 105-106, 108-109, 116-117, 120, 122
Classic Amiga 2, 18-19, 23, 49, 69, 89, 98, 114-115, 131, 137, 143-145, 148-149, 159, 161-162, 167, 181-182, 188, 197, 209-210, 231, 241, 261, 276-277, 279, 283-284
Classic IFF 277
Classname 280
clip a patch list 267
close button 258
closetoolkit 245, 260
clrtoken 244
cmenuitem 251, 259
Code Pseudo-ops 84
colour conversion 272
column 130, 251, 259, 268
command line 11-12, 57-58, 61, 64, 68-69, 71, 74, 126-127, 171-173, 175, 186, 192, 207-208, 210, 213, 221, 223-224, 226, 263, 265
Command Line Parameters 126, 171, 173, 186
Comms Events 255
compilation 36, 39, 46, 57, 61-62, 68-69, 71, 74, 94
Compilation Examples 62, 74
Compiler Options 68, 95
Compiling C 61
components of a class 280
compressed pixelmap 276-277
compressed soundmap 276-277
Conditions 27, 31, 67, 163, 220, 233
conformance 171-172
constant recompilation 39
containers 242
CONTINUE 26, 32, 139, 214, 240
CONTINUEIF 26, 32
conversion problems 66
Converting Classic IFF 277
Cooked key 255, 261
core class libraries 41
CP/M 23, 198, 201
CPM 276-277
CPU Isolation Interface 151, 177-178, 205
CPY 23-25, 31, 59, 100, 102, 106, 109, 111, 113, 115-122, 130-132, 134-137, 141, 145-146, 150, 185, 263-265
CSM 276-277
cut a region 267
Cyrillic 56

D

Data flow analysis 171-172

- Data hiding 281, 285
- Data Initialisation 83, 156
- data integrity 183
- data latency 23
- Data types 18, 36, 63-64, 285-286
- dbitstart 137, 263, 278
- dbx 86
- Deadline 157
- Deadline Monotonic Scheduling 157
- deallocation 99
- Debugger 3-4, 8, 10, 64, 86, 169-170, 178
- default return 93
- defaultmethod 99-101, 104, 106, 108-110, 116-117, 120, 122, 129, 184, 281
- Defining a Class 99
- deinitialisation 104
- Demonstration Programs 253
- denormalised numbers 18
- derefcass 103, 106, 109, 116, 118, 120, 122, 186
- design patterns 55
- destination path 91
- Developer Documentation 175
- Device Driver 7, 22, 65, 72, 75, 97, 152, 164, 167-168, 175, 180, 182-185, 187-193, 195-198, 203, 241, 243, 246-247, 252-253, 261, 272, 281
- device driver families 195
- devstart 167, 191-193
- DF_RESERVED 195
- diagnostic string 182
- Diagnostics Pseudo-ops 85
- dialog 251, 258, 266
- dialog window 258
- DIALOG_ACTION 258
- dingbats 56
- disassembler 3, 169
- dispatchk 243, 261-262, 264
- Displaying Audio Visual Objects 267
- double 13, 18-19, 23, 25, 28, 30-32, 63-64, 66-67, 92, 94, 130
- double precision 18-19, 23
- doubly-linked list 185
- DragToken 244
- Drawing Primitives 269
- Driver files 252

- driver instances 185
- Drop Event 255
- DSFX 241, 252, 272, 281
- Dynamic Binding 58, 90, 148, 151, 206, 281

E

- Earliest deadline First 157
- efficiency 40, 45, 90, 104, 131, 188, 221
- ellipses 269
- ELSEIF 32, 139
- Encapsulation 281, 285
- enclosed shapes 269
- entd 100-101, 106, 108-109, 116-117, 120, 122, 129
- entih 31, 129, 162
- entl 129
- entropy 152, 168
- environment list 49
- epilogue 58, 60
- equis.inc 144
- ermo 67, 81, 143, 193
- Error checking macros 143
- errorf 143-144
- Ethernet 195, 197
- event handler 248-249, 258, 266
- Event linking methods 257
- event type identifier 257-258
- Exceptions 36, 163, 224, 233-235
- Exec library 115
- existing applications 17, 66
- exit() 65, 127
- Expression Types 31

F

- Film playing and recording tools 253
- Fixed point 19, 31
- flavours of Unix 170, 197
- FLM 276-277
- float 31-32, 63, 66-67, 130, 134
- floating point 18-19, 26, 32, 282, 286
- focus 267
- Font 114, 241, 253, 274-275, 282, 286-287
- footprint 3, 11, 40-41, 48, 50, 206
- FOR count 26

fork 170
Format Conversions 32
format of listings 114
Format Qualifiers 32
fprintf 146
FPU 19, 282
fragile superclass 42
frame register 35
freeevent 245, 249
Front End 57
Function Call Conventions 63
Fundamental Algorithms 115

G

Gadget 242, 250-251, 253, 259-260, 266-268, 278, 285
gadget Audio Visual Object class 253, 259
Gadget Layout Tools 253
gadget toolkit 253, 259
GadTools 259
Gained token 255
gamma correction 274
garbage collection 33, 35, 39, 42, 45, 52
GCC bug 67
getevent timeout 249
getstate 259
gettoken 244
getvoice 272
GIF 276, 282, 286
global data initialisation 156
global data storage 58-59
Global Pointer 21, 150
Global Variable Size 125-126
Glyph 274, 282
go method 273
gose 72, 75, 84-85
Grammar 223, 236
Graphic Interchange File 276
Graphics Contexts 253
Graphics driver 241, 253, 282
Greek 56
GREP 10, 222
greyscale 272
GRF 195, 241, 252, 269, 282
GUI 15-16, 282

Gulliver's Travels 22
Guru Meditation 131

H

hackers 147
half-transparency 272
Han characters 56
handle 51, 57, 144, 181, 187, 191, 194, 235, 246-247, 249, 257, 269, 272-274
handler method 248-249
hardware I/O ports 183
Hayes 199
heap 34-35, 253, 267
Heap memory management 253
Hebrew 56
Hello 14, 33, 44, 91, 94-95, 123, 169, 225
hello.c 91
Heterogeneous 3, 151
heuristics 171
hex dump 22, 182
highest priority 157
Hints and Tips 175
hosted 1, 4, 169-170, 179-181, 197-198, 200-201
hosted thread 181
housekeeping 127
HTML files 175
htrans 272
HttpClassLoader 54

I

IEEE-754 18, 32, 282
IF condition 26
iferrno 111, 113, 143-144, 194
IFF 2, 277, 279, 283
Image 4, 6, 8, 44, 206, 242, 250, 253, 269, 272, 276-277, 280, 282-283
image Audio Visual Object class 253
indirection 24-25, 83
Inheritance 33, 37, 48, 250, 266, 283, 285
initflag 80-82
initialisation 59, 76, 78-81, 83, 88, 97, 99, 103-104, 110, 143, 156, 184, 222
initialisation routine 59
Input Method Tools 253
Input/Output 34, 36, 146, 183

- Input/Output package 36
- Inputtoken 244
- instance pointer 102, 106, 109-110, 167, 185, 193, 256
- instance variables 36-38, 186
- Instruction Types 31
- Instrumenting a tool 173-174
- int 22, 31-32, 59, 63, 65-67, 77, 92-96, 98, 115, 117-118, 134-136, 145, 150, 256, 263
- Inter Process Communication 151, 158
- intermediate codes 17
- intermediate file 61
- Internet 33, 41, 56, 199, 284
- Interrupt Handlers 65, 129, 162, 180, 183, 187
- Interrupt Management 181
- Interrupt Request 181, 184, 186-187
- Interrupt Service 181-182, 186-189, 191
- Interrupt Service Routine 181-182, 186-189, 191
- Introduction to Java 33
- introspection 55

J

- Japanese 288
- jar files 54
- Java archives 54
- Java Language 33-40, 42-44, 47, 54, 89, 279
- Java Language package 36
- Java Platform 34-35, 39, 41, 46
- Java program counter 35
- Java Virtual Machine 33-35, 41-42, 45-47, 51
- javac 15, 43, 50
- job 2, 23, 90, 110, 115-117, 121-122, 198
- JOVE 178, 220-222
- JPEG 283
- JPG 276, 283
- Just In Time 38, 46-47

K

- kernel variable 166
- Kerning 283
- key concepts 177

- Keyboard Events 243, 255
- Keypad 195
- Keytoken 244
- Knuth 115
- KN_ATOMLIST 166
- KN_ATOMTABLE 166
- KN_ATOMTEXT 166
- kprintf 143
- KTRACE 5, 26, 32, 143

L

- label 16, 27, 60, 84, 133, 138, 141, 144, 146, 246, 251, 259, 266
- Lao-Tsze 287
- Latency 23, 283
- Launch 44, 246
- Laxity 157
- lbutton 251, 259
- lcheckbox 251, 259
- linear PCM 272
- lines and points 269
- Link Pointer 21, 32, 150
- Linked list macros 144
- Linkonce blocks 78
- LINT 222
- Linux 1, 4, 7, 10, 69, 164, 169-170, 178, 210, 284
- List header 118, 144-145, 174
- List node 144-145, 166, 185
- little-endian 22
- log 104, 272
- logical colour mask 272
- logical input 247
- longjump 163
- Look and Feel Toolkits 253
- lookup 165, 167
- Lost token 244, 255

M

- Macros 20, 25-27, 30, 32, 86, 99, 124, 132, 134, 137-138, 140, 143-146, 183, 194
- magic string 79
- Mailbox 159, 243, 248, 261, 284
- Mailboxes 158-159, 161
- main AVE driver 253
- main event loop 254

- main() 34, 65
- mark phase 52-53
- masking 271
- Maximum Urgency First 157
- mbtowc 146
- MEMF_PUBLIC 161
- Memory Allocation 92, 102-103, 134, 160-161, 179-180
- memory efficiency 40, 104
- memory management 39, 151, 160-161, 180, 183, 253
- Memory Mapping 183
- memory objects 49, 160-161
- Memory Structure 98
- menu 16, 245-246, 251, 259
- menu item 16, 246, 259
- menubar 251, 259
- menuItem 251, 259
- message structure 255-256
- meta-instructions 25
- method area 34-35
- Method Coding 99
- method overloading 33
- microspacing 274
- Minimum Laxity First 157
- Mixing the trick 271
- modal root 249
- Modality 249
- Module Control 76
- monoshot 162
- Monotonic 157
- Motorola 17-19, 22-23, 25, 285
- mountlist 167
- mount_delayed 167
- Mouse button pressed 255
- Mouse button released 255
- Mouse entering 255
- Mouse Events 249, 255
- Mouse leaving 255
- Mouse tracking 255
- MP3 284
- MPEG 15, 284
- MSDOS 12, 197-198, 201
- mshade 272
- MTX_SIGMASK 163-164
- multi-tasking 42

- multi-threaded 90
- multimedia toolkit 14-16, 41, 44-45, 203-205, 241-242, 246-247, 252, 257, 267-268, 275, 279, 281-282
- multiple inheritance 33, 37
- Multiple modules 78
- Multiple options 11, 208
- Multithreading 3, 40, 42, 49
- mutex 158, 163-164, 183, 186
- mutexes 42, 158-159, 163, 186

N

- named data area 174
- named data areas 152, 165, 174
- named method call 284, 288
- naming convention 99, 198
- native code 3, 17, 41, 58, 89, 147, 178, 288
- natural boundaries 23
- nextid 245
- Node 60, 115-117, 119-120, 144-146, 166, 185, 285
- non-aligned 31, 131
- non-blocking 142, 158-159, 184, 188
- non-hosted 179
- Non-primary tools 124-125, 127
- noret 31, 99, 101, 169
- Not A Numbers 19
- notification 181, 188-189, 216
- Nuxi problem 22

O

- object orientated calls 29
- Object-based programming 41, 90, 97, 285
- Object-Oriented Programming 36, 41, 97
- odd addresses 23
- old C 66
- open method 194
- opentoolkit 245, 260
- operator overloading 33
- optimisation level 71, 75
- optimisations 21, 171-172
- optimiser 64
- optop 35
- optop register 35

- OS/9 197
- output tool name 73, 91
- ovals 269
- Overflow 19, 64, 169
- overhead 20, 23, 47, 64, 101, 160, 203, 270
- Overtoken 244
- P**
- Packages 38
- Packed Alignment 64
- Page Out Prevention 183
- parameter passing 92, 149, 169
- Parameter Pointer 21, 32, 149
- parentclass 99, 101, 108-109, 117, 122, 186
- passing a long 66
- paste a region 267
- PCALL 29, 108-109, 117, 121
- PCX 276
- PDA 285
- Per Process Data Block 58-59
- periodic timer 162
- persistence 55
- PersonalJava 14, 41, 45
- PF_CALLBACK_OCCURRED 165
- PID 156-157, 224, 286
- PII tools 179-180
- pipeline stalls 23
- Pixelmap 203, 242, 250-251, 253, 267-271, 276-279, 286
- pixelmap Audio Visual Object class 253
- Pixelmap conversion tools 253
- Pixelmap Types 251, 270
- plain fill 272
- platform independence 40, 151
- platform isolation interface 41, 151, 177-179, 205
- Plum Hall 47
- PNG 276, 286
- pointer parameters 66
- polygons 268-269
- Polymorphism 285-286
- portability 3, 29, 34, 41, 89, 147-148
- Portable Net Graphic 276
- Porting existing C code 66

- positional data 242
- POSIX 146, 156, 163, 178, 204, 213
- postlink 257-259
- PostScript 14, 241, 274, 286-287
- power management 181
- PPC 23, 125, 147-149, 169, 176, 285
- Pre-processing 61
- Pre-processor 57, 61-62
- PRINTF 26, 32, 93-94, 96, 124, 142-144, 146-147
- printf() 93
- priority 117, 155-157, 162, 191, 263
- Process control 156
- Process creation 155-156
- Process ID 157, 224, 227
- process table data 157
- processes 48-49, 58, 60, 93, 102, 152, 154, 157, 160-161, 165, 167, 188, 200, 224, 232, 237, 239, 245-246, 258
- programming the AVE 254
- prologue 58, 60
- pseudo-operations 25, 59-60, 64
- Pseudo-ops 76, 79-80, 82-88
- ptrace 170
- pure tools 79

Q

- Qdos 164
- QNX 161, 197
- Qualifiers 32, 130
- queue 115, 120-122
- Quick Reference 26, 31

R

- randomness 168
- Rate Monotonic 157
- Raw key down 255
- Raw key up 198, 255
- Raw keyboard 204, 261
- rbutton 251, 259
- rcheckbox 251, 259
- re-entrant 48, 90, 287
- read Method 187, 247
- reada 184, 189-190, 247
- real-time 40, 42, 151, 197
- refclass 102, 106, 109-110, 116, 118, 120,

- 122, 185
- Region patch tools 253
- Register Usage 63, 162
- Registers 18-21, 25, 29, 31-32, 34-35, 53, 63-64, 66, 80, 92, 128-130, 132-133, 136, 140-142, 149, 162, 170-171, 184
- Remote Method Invocation 50
- remove a region 267
- RenderWare 241
- REPEAT 26, 32, 75, 138, 230
- residual string 193-194
- Rexx 2
- rgbmask 272
- RISC 3, 17, 285, 287
- rlookup 167
- root directory 10, 51, 91, 123, 125, 143
- Round Robin 157
- RPM 276-277
- RSM 276-277

S

- salt solution 203
- sample format 272, 276
- scaling 271, 287
- scheduling 42, 49, 151-152, 155-157
- scheduling algorithms 151, 157
- scope 52, 58, 60
- screen size and mode 245
- scrollbar 251, 259
- scrollpane 251, 259
- secondary tool 95
- Sejin 193
- semaphores 42, 158-159
- servlet 55
- setjmp 163
- setstate 259
- settoken 244
- setvoice 273
- shadow 272
- Shell Commands 10, 12, 176, 207, 210, 213, 222-223, 225-226, 233-234, 239
- shell.argv 226
- shell.binpath 227
- shell.pid 224, 227
- short 2, 18-19, 23, 31-32, 40, 63, 66-67, 90, 130, 176-177, 208, 221, 245
- sign extension 20
- signal handlers 49
- signals 40, 42, 151, 157, 163-164, 188, 261, 284
- signature matching 48
- SIGUSRn 163
- SIG_DFL 163
- SIG_IGN 163
- single precision 18-19
- Size in bits 63
- size suffix 25
- SND soundmap class 241
- Softkey 246
- software interfaces 37
- sound Audio Visual Object class 253
- Sound device driver 253
- Soundmap 241-242, 250, 276-277
- SOX 277
- spawn structure 156
- Spawning 156
- stabs 64, 86
- stack frame 35
- Stack Pointer 21, 32, 63, 136, 149
- Stack Usage 64
- standalone executable tool 124
- standard gadget toolkit 253, 259
- standard message structure 255
- standard window toolkit 253
- Startup and Shutdown 180
- statics 49
- stdio 207
- stdout 127, 143, 171, 173-174, 232
- String drawing 274
- subclass 37, 98, 101, 103, 107-110, 112-114, 186, 250, 259, 266, 268, 281, 283
- subclassing 182, 250
- sublink 258
- submenus 16, 246
- subpatch 267
- suffix 18, 25, 31, 43-44, 57-58, 68-70, 72, 123, 148, 176, 218, 230
- supervisor 149, 163-164, 182
- suspend specified process 156
- sweep phase 53
- symbolic references 39
- synchronisation group 159

Synchronisation Groups 159
Synchronous 42, 184, 190, 247, 261-262, 264
syntax 10, 23, 33, 92, 139-142, 192, 209, 224, 232, 236-237, 239
system applications 243, 245-248, 252
System Events 39, 254
system menu 16, 245-246

T

Taiwanese 56
Tamil 56
Tao 1, 287
taort include file 104
taos_exit() 65
TCK test suite 47
TCP/IP 14, 195, 199-200
temporary files 61, 227
terminate and delete 156
textarea 251, 259
textfield 251, 259
TGA 277, 287
Thai 56
Thread 21, 39, 42, 48-49, 53, 125, 150, 154, 181, 287
threads 36, 42, 49, 154, 170, 188
Throwable 36
Tibetan 56
tidy up routine 103
Tidying up 13, 21, 127, 150
tiled backdrop 246
timeout 158-159, 249, 254
timer device 151
timeslicing 154
Tips for Programmers 175
title 251, 259
togrey 272
token allocation 244
Token Events 255
token types 244
Tool Generation 64
tool loader 152-153, 175-176, 198
tool name macro 125
tool names 125
tooltype 167
tracef 32, 64, 100, 104-106, 109, 111-113,

116, 119-120, 142-144, 150
tracef() 64
Translation 3, 34, 39-45, 50-52, 61, 89, 153, 176
translator 3, 5, 17-18, 20-26, 28, 44-45, 52, 57-58, 61-62, 73, 76, 89, 93, 131, 149, 152, 169, 176, 198, 205-206
translucency 272
translucent 269
transparency 269, 278-279
TrueType 14, 241, 274, 287
Typeface 282, 286, 288

U

unaligned 131, 134
Unicode 56, 204, 288
unlog 273
unreachable code 68, 171
unsetexc 181
unsuspend specified process 156
updatepatches 267
Urgency 157
User List node 145
user mode 163, 182
user.locale 227
user.tmp 227
UTF-8 56
Utility Package 36

V

variable arguments 93
variable-length argument 66
variable-length argument lists 66
vars register 35
vgrad 272
Virtual Machine 26, 33-35, 41-42, 45-47, 49, 51, 89, 147
Virtual Processor 3, 17-18, 41-42, 45, 47, 89, 128, 149, 262, 287-288
VLIW 3, 288
VP conformance checks 172
VP Processor Model 32
vpas 57-58, 69, 72, 74, 80, 84, 86, 88
vpcc 57, 61-62, 65, 68-69, 74-75, 91, 95
vpcc.log 65
VPcode 1, 3, 17-18, 20, 22-30, 90, 125,

148, 151, 182, 205, 288

vp_init() 59

VT52 203

W

Wait for a child 155-156

waitpid 170

wake specified process 156

warning options 66

WAV 195, 276, 288

weak property 82

Web connections 56

Window 7, 10, 36, 44, 222, 241-242,
250-251, 253, 258

window Audio Visual Object class 253

window toolkit 36, 44, 253

World Wide Web 33, 46

wrotea 189-190

X

XBM 276

xmethod 101, 270, 288

XYZ co-ordinate 242

Z

z plane 242

zapping 20, 68, 71

—

_deinit 100, 103-104, 106, 108-110, 115,
117, 119, 121, 184, 186

_deinit Method 100, 103-104, 108, 110,
186

_delete 99, 102-103, 106, 110, 185

_init 9, 100, 103-105, 108-109, 115,
117-118, 121, 184, 186, 192

_init Method 100, 103-104, 108, 186, 192

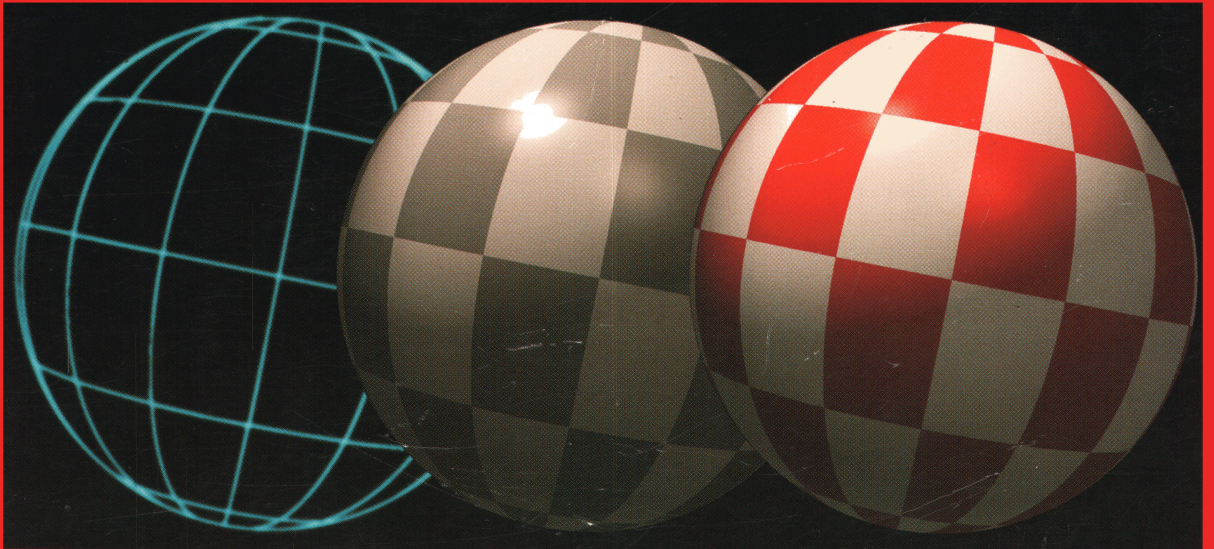
_new 99, 101-103, 106, 110, 185, 192

__attribute__ 65, 92-96

__GNUC__ 73

__OPTIMIZE__ 71

__TAOS_TOOL__ 72



Amiga SDK 2000



**This was brought to you
from the archives of**

<http://retro-commodore.eu>